A GROUP THEORETIC TABU SEARCH APPROACH TO THE TRAVELING SALESMAN PROBLEM

THESIS

Shane N. Hall, First Lieutenant, USAF

AFIT/GOR/ENS/00M-14

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE

A GROUP THEORETIC TABU SEARCH APPROACH TO THE TRAVELING SALESMAN PROBLEM

5. FUNDING NUMBERS

6. AUTHOR(S)

Shane N. Hall, First Lieutenant, USAF

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GOR/ENS/00M-14

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ AMC/XPY
402 Scott Drive, Unit 3L3
Scott AFB, IL 62225-5307
DSN: 576-5954

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Dr. James T. Moore, AFIT/ENS (James.Moore@afit.af.mil) (937) 255-6565

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

ABSTRACT (*Maximum 200 Words*)

The traveling salesman problem (TSP) is a combinatorial optimization problem that is mathematically modeled as a binary integer program. The TSP is a very important problem for the operations research academician and practitioner. This research demonstrates a Group Theoretic Tabu Search (GTTS) Java algorithm for the TSP. The tabu search metaheuristic continuously finds near-optimal solutions to the TSP under various different implementations. Algebraic group theory offers a more formal mathematical setting to study the TSP providing a theoretical foundation for describing tabu search. Specifically, this thesis uses the Symmetric Group on n letters, S_n, which is the set of all $n!$ permutations on n letters whose binary operation is permutation multiplication, to describe the TSP solution space. Thus, the TSP is studied as a permutation problem rather than an integer program by applying the principles of group theory to define the tabu search move and neighborhood structure. The group theoretic concept of conjugation (an operation involving two group elements) simplifies the move definition as well as the intensification and diversification strategies. Conjugation in GTTS diversifies the search by allowing large rearrangement moves within a tour in a single move operation. Empirical results are presented along with the theoretical motivations for the research.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Group Theory, Tabu Search, Heuristics, Metaheuritics, Traveling Salesman Problem, Java, Permuations	109
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNC

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GOR/ENS/00M-14

A GROUP THEORETIC TABU SEARCH APPROACH TO THE TRAVELING SALESMAN PROBLEM

THESIS

Presented to the Faculty

Department of Operational Sciences

Graduate School of Engineering and Management

of the Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Operations Research

Shane N. Hall, B.S.
First Lieutenant, USAF

March 2000

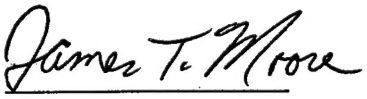
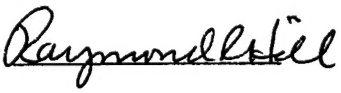
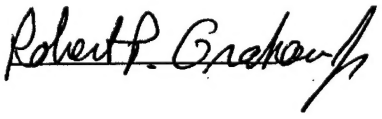
AFIT/GOR/ENS/00M-14

THESIS APPROVAL

NAME: Shane N. Hall, First Lieutenant, USAF **CLASS**: GOR-00M

THESIS TITLE: A Group Theoretic Tabu Search Approach to the Traveling Salesman Problem

DEFENSE DATE: 24 February 2000

COMMITTEE:	**NAME/TITLE/DEPARTMENT**	**SIGNATURE**
Co-Advisor	James T. Moore, Ph.D. Associate Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology	
Co-Advisor	Raymond R. Hill, Major, USAF Assistant Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology	
Reader	Robert P. Graham Jr., Major, USAF Assistant Professor of Computer Engineering Department of Electrical and Computer Engineering Air Force Institute of Technology	

Acknowledgements

I wish to express my sincere thanks for several individuals who have guided me throughout my research. First, thanks to Lieutenant Colonel T. Glenn Bailey, now retired, who introduced me to the concept of group theoretic tabu search. His motivation and enthusiasm was an inspiration to work hard and strive for the best. Next, thanks to Dr. James T. Moore for his guidance and interest in my research. I have great respect for his patience, style, and knowledge. I also express thanks to Lieutenant Colonel (sel) Raymond Hill and Major Robert Graham for their efforts in wading through my research and providing useful comments and suggestions. Finally, I thank Dr. Bruce Colletti and Dr. J. Wesley Barnes for their motivation, enthusiasm, and inspiration. I honor and respect each of these professors and will strive to serve with their same qualities.

Finally, my deep gratitude for the special heroes. To my wife, Camalee, and my two sons, Nikolaus and Ammon, I express my love, respect, and gratitude for their sacrifice. Only they know and understand the sacrifices and events of the last eighteen months. To my father and mother, who taught me principles and truths much greater than tabu search and group theory, I express my honor, love, and respect.

Shane N. Hall

Table of Contents

List of Figures

List of Tables

Abstract

The traveling salesman problem (TSP) is a combinatorial optimization problem that is mathematically modeled as a binary integer program. The TSP is a very important problem for the operations research academician and practitioner. This research demonstrates a Group Theoretic Tabu Search (GTTS) Java algorithm for the TSP. The tabu search metaheuristic continuously finds near-optimal solutions to the TSP under various different implementations. Algebraic group theory offers a more formal mathematical setting to study the TSP providing a theoretical foundation for describing tabu search. Specifically, this thesis uses the Symmetric Group on n letters, S_n, which is the set of all $n!$ permutations on n letters whose binary operation is permutation multiplication, to describe the TSP solution space. Thus, the TSP is studied as a permutation problem rather than an integer program by applying the principles of group theory to define the tabu search move and neighborhood structure. The group theoretic concept of conjugation (an operation involving two group elements) simplifies the move definition as well as the intensification and diversification strategies. Conjugation in GTTS diversifies the search by allowing large rearrangement moves within a tour in a single move operation. Empirical results are presented along with the theoretical motivations for the research.

Chapter 1. Introduction

1.1 Background

Optimization is a vast and important area of study in the discipline of operations research. Optimization problems are encountered on a daily basis in military and industrial operations, particularly in the areas of manufacturing and transportation. The basis for many such manufacturing and transportation problems is the idea of selecting an optimal, or perhaps feasible, way to partition or order jobs, vehicles, locations, etc. Such problems fall into the class of combinatorial optimization. In reality, most of the real world problems in combinatorial optimization are extremely large and complex, but they nonetheless require timely solutions. Much research in classical optimization and heuristic methods has been performed to find good and timely solutions to these large real world problems. This research studies the specific combinatorial optimization problem known as the traveling salesman problem (TSP). This research studies the TSP in the light of algebraic group theory and the tabu search metaheuristic. This chapter contains a general description of the problem, defines the research scope and contributions, and gives an overview of this thesis.

1.2 Problem Description

The classic TSP requires an agent to leave his base location and visit several other customer locations exactly once before returning home. The solution specifies the order in which the locations should be visited and we wish to minimize/maximize some objective, such as the total distance traveled. The dimensionality of this ordering grows

exponentially as the number of locations increase. As an example, a relatively small TSP with 40 locations has 40!, which is more than 8×10^{47}, possible orderings . Even with modern computing power, these problems quickly become overwhelmingly large.

If a visit to a location depends on the arrival or departure time, or both, then we have a TSP with time windows (TSPTW). A mTSP is described in the same manner where there are multiple agents that may visit the desired locations, instead of only one agent. Likewise, if a location must be visited within some interval of time, then we have a mTSPTW.

The mTSPTW arises daily in many real world situations. Air Mobility Command (AMC) encounters a complex variation of the mTSPTW in its daily operations of airlift and refueling. In the case of AMC, the "salesmen", or agents, are airlift or tanker aircraft that must visit each of the required drop off or refueling locations with the objective of maximizing throughput or minimizing cost. AMC's problem is more complex in that the salesmen, i.e., aircraft, have limiting cargo capacities that must satisfy all customer demands at each location. This more complex problem is the vehicle routing problem with time windows (VRPTW).

Air Mobility Command's airlift operations are simulated by the Mobility Analysis Support System (MASS) model. Aircraft routes are an input to this low-resolution model, which then analyzes capabilities and assesses future procurements. AMC seeks more efficient ways to generate quality routes for its operations and simulations.

1.3 Scope

This research seeks a more efficient way to find solutions to the routing problems encountered by AMC using group theory, an important branch of abstract algebra, and the tabu search metaheuristic. A group is an algebraic structure that ensures unique solutions to simple algebraic equations. This research uses the group of permutations on n letters known as the Symmetric Group on n letters, S_n. Conveniently, in the study of the TSP each solution is a unique element of S_n. The basic concepts of group theory needed for this research are presented in Chapter 2.

Tabu search is a metaheuristic procedure that has proven very effective in large combinatorial optimization problems. Tabu search uses *recency* (short-term memory) and *frequency* (long-term memory) to search a solution space in an intelligent and efficient way. Recency and frequency are used to avoid cycling and to escape local optimum by diversifying into unvisited, and perhaps infeasible, regions of the solution space.

This research incorporates algebraic group theory to define the tabu search method used to solve the TSP, which is the base problem to AMC's airlift routing problem. The group theoretic tabu search method (GTTS) is implemented in the Java programming language.

1.4 Contribution of Research

The mathematical formulation of the TSP is well known, and many algorithms, heuristics, and software packages that solve this mathematical formulation exist. Various tabu search methods have been applied to the TSP; however, a GTTS approach for the

TSP does not exist. The general concepts of tabu search applied to combinatorial optimization are an active area of research and, until recently, have not been developed with mathematical rigor. The GTTS method seeks to demonstrate a tabu search method for the TSP in a mathematically rigorous way. Thus, the GTTS method is described with algebraic equations rather than narrative.

Once tabu search concepts are fully developed mathematically, it is probable that more efficient algorithms will be developed which give near optimal solutions to extremely large combinatorial optimization problems such as those faced by AMC. If nothing else, understanding the mathematical foundations of tabu search will provide needed insight into the behavior of the tabu search metaheuristic.

1.5 Overview

This thesis comprises five chapters and two appendices. Chapter 2 describes the associated current literature and the basic concepts of group theory needed for this research. Chapter 3 describes the methodology for solving the TSP with GTTS. The results of this methodology are presented in Chapter 4. Finally, Chapter 5 concludes the research and discusses future research. Appendix A contains the necessary Java documentation and source code for the GTTS algorithm, and Appendix B presents a detailed mathematical model for the TSP and its related class of problems.

Chapter 2. Literature Review

2.1 Introduction

This chapter describes the foundational literature for this research. The three main areas of interest are the TSP class of problems, heuristic search methods, and group theoretic metaheuristics.

2.2 The Traveling Salesman Problems

The TSP is the focus problem for this research. The TSP encompasses a whole class of problems such as the mTSPTW, the VRP, and the pick-up and delivery problem (PDP). This section reviews the TSP literature relevant to this research. Bodin *et al.* (1983) covers in detail the TSP and its class of problems where each problem's background and formulation is given. Much of the TSP literature used for this research is taken from Ryer's (1999) application of reactive tabu search to the TSP. Ryer's formulation of the TSP and the more complex problems derived from it in turn relies heavily on the work of Bodin *et al.* (1983) and Carlton (1995). Ryer expands Carlton's work to formulate the multiple depot VRP with non-homogeneous vehicles. The literature just presented provides formulations for the TSP class of problems as binary integer programs. See Appendix B for these formulations.

2.3 Heuristic Search Methods

Most real world combinatorial optimizations problems, such as those discussed in Section 2.2, are extremely large and complex. Such problems are classified as NP-hard

combinatorial problems (Nemhauser and Wolsey, 1988), which implies that the number of solutions to the problem grows exponentially as the number of locations, or nodes, grows. As a simple illustration, consider the 6 node TSP and the 7 node TSP. There are 120 distinct solutions to the 6 node TSP verses 720 distinct solutions to the 7 node TSP. NP-hard problems have no polynomially bounded algorithm (Baker and Schaffer 1986). However, to reach the optimal solution in a reasonable amount of time requires a polynomially bounded algorithm. Thus, these problems must be attacked using heuristic search methods that produce good solutions in a timely manner. Although heuristics do not guarantee optimality, the efficiency and near-optimal solution quality of many heuristic search methods justify their use in solving large combinatorial optimization problems.

The literature contains several heuristic search methods for solving the TSP class of problems. Laporte (1992a, 1992b) discusses many heuristic methods for the TSP class of problems, and classifies heuristic search methods for the TSP as either tour construction algorithms or tour improvement algorithms. Tour construction algorithms, such as the nearest neighbor heuristic and nearest insertion heuristic (Nemhauser and Wolsey, 1988) pick some initial node and construct a tour based on certain criteria. Tour improvement algorithms include the *k*-opt algorithms where *k* arcs are dropped from an existing tour and then *k* new arcs are added back to form an improving tour. This is done iteratively until no improvements may be found. For example, consider the TSP tour: begin at node 0 → node 1 → node 2 → node 3 → back to node 0, denoted (0,1,2,3). A 3-opt algorithm may be defined as a swap of any two adjacent nodes within the tour.

Hence, the new tour (1,0,2,3) may result from one iteration. Here, we assume that the arc *a-b* is different from arc *b-a*.

Familiar heuristics such as greedy algorithms, genetic algorithms, simulated annealing, and tabu search are more sophisticated search methods that include tour construction and tour improvement. Greedy algorithms are useful for simpler problems but do not provide the desired solution quality. Simulated annealing shows a large variance in solution quality and computational time (Osman 1993). Tabu search has proven to be very effective at solving the TSP class of problems, even outperforming other sophisticated heuristics (Carlton 1995, Glover and Laguna 1997, Laporte 1992b, Ryer 1999).

2.3.1 Tabu Search

Tabu search (TS) is a metaheuristic search method originated by Glover (1990) that has proven very effective in solving large combinatorial optimization problems. TS uses *recency* (short-term memory) and *frequency* (long-term memory) to search a solution space efficiently by allowing the search to leave local optima and search other areas of the solution space. TS examines a given solution's *neighborhood*, which is defined to be the set of all moves from the given solution to a new solution. The *move* is problem specific and may change as the search progresses. For example, a move definition for the TSP may be the 3-opt move described above where any two adjacent nodes in the given solution are swapped. Another example of a TS move is toggling a binary variable in an integer programming problem from zero to one.

TS utilizes recency (short-term memory) by remembering its most recent moves and classifying them as "tabu" or forbidden. For example, the search would classify the nodes swapped in the 3-opt TSP move as tabu for a certain number of iterations. The number of iterations a move remains tabu is called its *tabu tenure*. This keeps the search from revisiting previous solutions, thereby preventing short-term cycles. Moves that are tabu may still be performed if they pass an *aspiration criteria*. An aspiration criteria used frequently in the literature is to allow a tabu move if the move results in the best solution encountered to date.

Tabu search uses frequency (long-term memory) for *intensification* and *diversification* purposes. Recording the number of iterations that each node is moved is a TSP example of frequency. Intensification uses frequency to determine historically good solutions and then intensifies the search in these regions. For example, suppose that the search history indicates that placing node 1 after node 3 provides good solutions. The search may intensify by only picking moves that enforce that condition. Diversification uses frequency to determine regions in the solution space not yet visited and then diversifies the search into these regions. For example, suppose the search history has rarely seen node 5 before node 1; the search may diversify by making moves that place node 5 before node 1 (Glover and Laguna, 1997).

TS is a broad problem solving methodology and not a strigently defined algorithm such as the simplex method in linear programming. Hence, TS must be tailored to the particular problem type with unique stipulations for such things as neighborhood construction and intensification/diversification strategies. Glover and Laguna (1997)

presents several different TS strategies depending on the problem type. An important goal of this research is the development of a GTTS algorithm for a general TSP.

2.3.2 Reactive Tabu Search

Battiti and Tecchiolli (1994) present a reactive TS scheme (RTS) that implements the basic TS methodology but allows the tabu tenure to react based on recent search history, i.e., the tabu tenure is adjusted according to the quality of the search. As an example, consider a TS problem with tabu tenure of three iterations. If the search cycles to a previous solution within a specified number of iterations, then the search reacts by increasing tabu tenure to five iterations. Conversely, tabu tenure is adjusted to two iterations if the search does not visit a previous solution.

Ryer (1999) implements a RTS Java algorithm for the mTSP that extends Carlton's (1995) code through Ryan *et al.*'s (1999) MODSIM implementation. Ryer uses both swap and insertion moves to define the solution neighborhood and tracks redundant tours with a two-attribute hashing scheme. The first attribute is the objective function value of the tour, and the second attribute is the tour hashing value described by Woodruff and Zemel (1993). The hashing scheme's purpose is to minimize the number of collisions (identifying two distinct solutions as identical). See Carlton (1995) for information on RTS applied to the TSPTW, mTSPTW, and the VRPTW.

2.4 Group Theoretic Metaheuristics

Metaheuristics is an extensive area of research within operations research; however, group theoretic metaheuristics is still in its infancy. Until recently, group theory has been applied in combinatorial optimization only in exact methods but not

explicitly to metaheuristic methods (Colletti, 1999; Colletti and Barnes, 1999). Colletti (1999) explores the explicit application of group theory to metaheuristic methods, and demonstrates how to apply tabu search to the TSP and mTSP. This research uses this literature as its foundation for group theoretic metaheuristics concepts and employs a tabu search algorithm that explores the usefulness of these concepts. The following sections present the basic concepts of group theory used in this research. An excellent reference for these concepts is Fraleigh (1994) or Colletti (1999).

2.4.1 Groups

A group is an axiomatic algebraic structure that ensures unique solutions to simple algebraic equations. For example, $5 + x = 8$ or $3x = 24$ have the unique solution x because their algebraic structure satisfies the group axioms. Thus, group theory allows the formulation of simple equations. This is an important contribution to metaheuristics since the current format of metaheuristic literature is narrative rather than mathematical (Colletti, 1999). Some formal definitions follow.

> Definition: A **group**, denoted $\langle G,* \rangle$, is a set G together with an associative binary operation * defined on G such that G has an identity element and each $g \in G$ has an inverse.
>
> Definition: A **binary operation**, *, is a rule that assigns each ordered pair (g, h), $g, h \in G$, to a unique element also in G. This implies * is closed on G.
>
> Definition: A binary operation, *, is **associative** if $a * (b * c) = (a * b) * c$, $\forall$ a, b, $c \in G$.
>
> Definition: An **identity** is an element $e \in G$ such that $g * e = e * g = g$, $\forall g \in G$.
>
> Definition: An **inverse** for $g \in G$ is an element $g^{-1} \in G$ such that $g * g^{-1} = g^{-1} * g = e$.

Definition: Let $n \in Z$, $g \in G$, then $g^n = \underbrace{g * g * g * \cdots g}_{n \text{ times}}$ is the n^{th} **power** of g. It is true that $g^n * g^m = g^{n+m}$ for all $n, m \in Z$. Also, $g^0 = e$ and $g^n = (g^{-1})^{|n|}$ for $n < 0$. (Colletti, 1999)

Definition: The **order of g**, $g \in G$, is the minimum n, $n > 0$, such that $g^n = e$.

Therefore, a group is any set together with a binary operation that satisfies the axioms of associativity, identity, and inverse. It can be shown that the identity element is unique and that each element in the group has a unique inverse. For notational simplicity, a group is often represented by its set of elements, such as G; however, any group must have both a set of elements and an associated binary operation. For example, the group of integers, Z, does not make sense unless an associated binary operation is understood. (Fraleigh, 1994)

Examples

Three groups are used to illustrate the important concepts of group theory. The first is a familiar group encountered in the first year of grade school. This is the group of integers under addition, i.e., <Z,+>. The second group is the simple finite group $<Z_4,+>$ where Z_4 = Z modulus 4 = {0, 1, 2, 3}. The third group, the group of permutations on n letters under permutation multiplication, is not as familiar as the first two, but it is the primary group used for this research. This group is called the symmetric group on n letters, S_n. This section introduces abstract concepts of group theory, illustrates these concepts with the familiar group <Z,+> and the simple finite group $<Z_4,+>$, and then demonstrates the concepts on S_n. Let us consider the formal concepts stated above.

- $\langle Z,+\rangle$: Addition is an associative binary operation on the set of integers, i.e. take any two integers, say x and y, then $x+y=z\in Z$ and $x+(y+z)=(x+y)+z$. Letting $x \in Z$, then zero is the identity element since $0+x=x+0=x$, and $-x$ is the inverse of x since $-x+x=x+-x=0$. For the group element 2, the 4th power, $2^4=2+2+2+2=8$ and the order of 2 is infinite. For contrast, the integers under multiplication, $\langle Z,\times\rangle$ is not a group because 2 does not have an inverse in Z; that is, there is no integer x such that $2x=x2=1$.
- $\langle Z_4,+\rangle$: Addition is associative and zero is the identity. Each element has an inverse, i.e., $0+0=0$, $1+3=3+1=0$, and $2+2=0$. For the group element 1, the 4th power is $1^4=1+1+1+1=0$ and the order of 1 is 4 since 4 is the minimum $n>0$ such that $g^n=1^4=0=e$.

2.4.2 The Symmetric Group, S_n

The symmetric group on n letters is the group that explicitly applies to this research.

> Definition: Let A and B be two sets and let f be a function such that $f:A\to B$. f is **one to one** (1-1) if each $b\in B$ has at most one $a\in A$ mapped into it, that is, for $x,y\in A$, $f(x)=f(y)$ implies $x=y$. f is **onto** if every $b\in B$ has at least one $a\in A$ mapped into it.
>
> Definition: Let A be a set; then a **permutation of a set** A is a function ϕ such that $\phi: A \xrightarrow{\text{1-1 and } onto} A$, or ϕ is a 1-1 function of A onto A.
>
> Definition: Let $A=\{1,2,\ldots,n\}$. The group of all permutations of A is the **symmetric group on n letters**, denoted S_n. (Fraleigh, 1994)

S_n is important because the solutions to the TSP class of problems are permutations! Group theory provides an alternative mathematical view to these

problems. Instead of viewing the solutions as large binary **x** vectors, they may be viewed as permutations on n nodes, i.e., elements of S_n. An element of S_n may be expressed as a *standard form permutation* which is a 2 by n matrix whose top row is the ordered set A, and the bottom row is the image of A under the function ϕ (Colletti, 1999). For example, let p, q $\in S_4$, where

$$p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \text{ and } q = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}.$$

Here, p and q are 1-1 and onto functions that map the positions into the letters. For example, p maps 2 into the first position ((1)p = 2) and q maps 1 into the third position ((3)q = 1). Elements of S_n may also be expressed in cyclic form. The cyclic representation of p and q above are p = (1,2,3)(4) and q = (1,4,3)(2). The cyclic representation is more compact and is explained below.

S_n is defined as a group, but this implies S_n must have an associative binary operation with an identity element and an inverse for each element in S_n. The associative binary operation for the group is permutation multiplication, or function composition, and is known to be associative. Consider p, q $\in S_4$ above. The product of p and q, $(x)(p * q)$ (or $(x)pq$ for notational ease) is the composition $((x)p)q$. Therefore (1)pq = ((1)p)q = (2)q = 2, pq(2) = (p(2))q = (3)q = 1, and so forth to yield p * q = pq =

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}.$$

Observe that $qp = \begin{pmatrix}1 & 2 & 3 & 4\\ 4 & 2 & 1 & 3\end{pmatrix}\begin{pmatrix}1 & 2 & 3 & 4\\ 2 & 3 & 1 & 4\end{pmatrix} = \begin{pmatrix}1 & 2 & 3 & 4\\ 4 & 3 & 2 & 1\end{pmatrix}$. Further $pq \neq qp$, i.e., permutation multiplication is generally not commutative, implying S_n need not be *abelian* (a group is abelian if * is commutative, that is, $p * q = q * p$ for all $p, q \in G$).

The identity and inverse properties for S_n are relatively straightforward. The permutation $( ) = \begin{pmatrix}1 & 2 & 3 & \dots & n\\ 1 & 2 & 3 & \dots & n\end{pmatrix}$ is the identity element for S_n. For example, $( )p =$

$$p( ) = \begin{pmatrix}1 & 2 & 3 & 4\\ 1 & 2 & 3 & 4\end{pmatrix}\begin{pmatrix}1 & 2 & 3 & 4\\ 2 & 3 & 1 & 4\end{pmatrix} = \begin{pmatrix}1 & 2 & 3 & 4\\ 2 & 3 & 1 & 4\end{pmatrix}\begin{pmatrix}1 & 2 & 3 & 4\\ 1 & 2 & 3 & 4\end{pmatrix} = \begin{pmatrix}1 & 2 & 3 & 4\\ 2 & 3 & 1 & 4\end{pmatrix} = p.$$

The inverse for any $p \in S_n$ is found by swapping its rows and then reordering the columns such that the top row is the ordered set A. Hence, for p, we have $p^{-1} =$

$\begin{pmatrix}2 & 3 & 1 & 4\\ 1 & 2 & 3 & 4\end{pmatrix} = \begin{pmatrix}1 & 2 & 3 & 4\\ 3 & 1 & 2 & 4\end{pmatrix}$. Note $p^{-1}p = pp^{-1} = ( )$ as required.

Recall that $p = \begin{pmatrix}1 & 2 & 3 & 4\\ 2 & 3 & 1 & 4\end{pmatrix} = (1,2,3)(4)$ in its cyclic representation. It is read $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $4 \rightarrow 4$. Cycles are the disjoint subtours of the permutation. Colletti (1999) defines an ***m*-cycle** (cycle) as an ordered sequence of m letters (whose order is precisely m). We define an ***m*-cycle** (cycle) as any sequence of m distinct letters whose order is m. Under this definition the 3-cycle $(1,2,3) = (2,3,1) = (3,1,2)$. Hence, the cycle need not be ordered with respect to the set $A = \{1, 2, \dots, n\}$. A more rigorous definition for a cycle is given by Fraleigh (1994), but requires additional concepts of group theory beyond the scope of this thesis. Henceforth, the cyclic representation for permutations is used.

Since S_n is a group, it is closed under multiplication, i.e., any product of permutations yields another permutation. Hence, cycles are permutations and may be multiplied with other cycles to yield other permutations. For example, p, above is the product of two disjoint cycles (a 3-cycle and a 1-cycle) as expressed above. We say disjoint because neither cycle has a common element. This illustrates the following theorem.

> Theorem 1: Every permutation of a finite set $A = \{1, 2, \ldots, n\}$ is a unique product of disjoint cycles. (Fraleigh, 1994)

Disjoint permutations yield a very useful property: they commute. The permutations p and q above are expressed by cycles p = (1,2,3) and q = (1,4,3). The 1-cycles are implied; however, it is important to know the symmetric group to which p and q belong. For example, p = (1,2,3) $\in S_4 \neq$ q = (1,2,3) $\in S_5$ since p = (1,2,3)(4) and q = (1,2,3)(4)(5). Since p and q are not disjoint cycles, it is not necessary that they commute. This is consistent with our earlier calculation that pq $\neq$ qp.

Finally, consider p = (1,2,3) $\in S_4$. The 3rd power of p is p^3 = (1,2,3)(1,2,3)(1,2,3) = (1)(2)(3)(4) = ( ) implying that p is of order 3. This is consistent with the definition of an m-cycle, which is defined to be of order m.

2.4.3 Subgroups

We see that the solutions to the TSP class of problems are elements of the group S_n. Although S_n is finite, the solution space can be extremely large. In some cases, it is possible to reduce the solution space, S_n, to a "smaller" space that shares the same algebraic structure, namely, that of a group. A "smaller" solution space is one with fewer

elements in the group set. The word "subgroup" intuitively suggests a group within a group.

> Definition: Let H be a subset of the group G, denoted $H \subseteq G$. If H is also a group, then H is a **subgroup** of G, denoted $H \leq G$.
>
> Definition: Let $g \in G$, then $H = \{g^n \mid n \in Z\}$ is the **cyclic subgroup of** G **generated by** g, denoted $\langle g \rangle$.
>
> Definition: A group G is **cyclic** if there exists some $g \in G$ such that $\langle g \rangle = G$. The element g is said to **generate** G.
>
> Definition: A 2-cycle permutation is a **transposition**.
>
> Definition: A finite permutation is **even** if it can be expressed as a product of an even number of transpositions; otherwise, it is an **odd** permutation.
>
> Definition: The group of all even permutations on n letters is the **alternating group on** n **letters**, denoted A_n.

It can be shown that a subset H is a subgroup of a group G if H is closed under *, $e \in H$, and $h^{-1} \in H$ for all $h \in H$. Thus $\langle 2Z,+\rangle \leq \langle Z,+\rangle$, where $2Z = \{\ldots, -4, -2, 0, 2, 4, \ldots\}$, and $\{0, 2\} \leq \langle Z_4,+\rangle$. However, the odd integers (together with zero) under addition do not form a subgroup of $\langle Z,+\rangle$. Although the subset $H = Z_{odd}$ contains the identity and each element has an inverse, H is not closed under addition; i.e., $1 + 3 = 4 \notin Z_{odd}$. Likewise the set $H = \{0, 1\} \subseteq Z_4$ is not a subgroup because H is not closed ($1 + 1 = 2 \notin H$) and each element in H does not have an inverse (namely, the element 1).

Interestingly, the order of g for some $g \in G$ discussed above equals the order or cardinality (number of elements) of the minimum subgroup H containing g. For example, the order of $1 \in Z_4$ is 4. Hence, the smallest subgroup of Z_4 containing the element 1 must have at least 4 elements implying that the smallest subgroup containing 1 is all of Z_4. Conversely, the order of $2 \in Z_4$ is 2 and $H = \{0, 2\} \leq \langle Z_4,+\rangle$. Clearly, $H =$

<2> is the cyclic subgroup of Z_4 generated by 2. In addition, <1> = $\{1^n \mid n \in Z\}$ = {. . ., $-1^2, -1^1, 1^0, 1^1, 1^2, \ldots\}$ = {. . ., -2, -1, 0, 1, 2, . . .} = <Z,+> implying <Z,+> is cyclic. Similarly, $<Z_4,+>$ is cyclic since <1> = <3> = $<Z_4,+>$. In S_4, H = { ( ), (1,2,3), (1,3,2)} is a subgroup. Recall the order of p = (1,2,3) is 3, which is precisely the cardinality of the subgroup H containing p. Also, H = <p> = <(1,2,3)> = { ( ), (1,2,3), (1,3,2) } is the cyclic subgroup of S_4 generated by p.

An interesting subgroup of S_n is A_n as defined above. A_n demonstrates the usefulness of subgroups. For illustration, the feasible solutions to a TSP lie in A_n provided n is odd. Since A_n is a group, it shares the same group structure as S_n but is a fraction (actually one half) of the size.

A_n is a group of even permutations, meaning every permutation in A_n may be expressed as a product of an even number of transpositions. The factorization of a permutation into transpositions is not unique. Consider the cycle (a,b,c,d). Expressed as transpositions (a,b,c,d) = (a,b)(a,c)(a,d) = (d,a)(d,b)(d,c). Although the factorization into transpositions is not unique, the cycle (a,b,c,d) is always a product of three transpositions, which implies any 4-cycle is an odd permutation. In general, any m-cycle is an odd permutation if m is even and is an even permutation if m is odd.

2.4.4 Cosets

In addition to possibly reducing the size of the solution space, subgroups partition groups into either identical or disjoint cells. Hence, the solution space S_n for the TSP class of problems may be partitioned into disjoint cells. These cells are called cosets and are useful in representing certain tabu search move methods (Colletti, 1999).

Definition: Let $H \leq G$. The subset $gH = \{ g * h = gh \mid h \in H \}$ is the **left coset** of H containing g. Similarly, $Hg = \{ h * g = hg \mid h \in H \}$ is the **right coset** of H containing g.

Definition: Let $H \leq G$. The number of left (right) cosets of H in G is the **index** $(G:H)$ **of** H **in** G.

Consider the group $\langle Z,+ \rangle$. Let $H = \langle 2Z,+ \rangle$, then the sets

$$0 + 2Z = \{ \ldots, 0 + -4, 0 + -2, 0 + 0, 0 + 2, 0 + 4, \ldots \} = \{ \ldots, -4, -2, 0, 2, 4, \ldots \} = 2Z;$$

$$1 + 2Z = \{ \ldots, 1 + -4, 1 + -2, 1 + 0, 1 + 2, 1 + 4, \ldots \} = \{ \ldots, -3, -1, 1, 3, 5, \ldots \} = Z_{odd};$$

$$2 + 2Z = \{ \ldots, 2 + -4, 2 + -2, 2 + 0, 2 + 2, 2 + 4, \ldots \} = \{ \ldots, -2, 0, 2, 4, 6, \ldots \} = 2Z;$$

are left cosets of H. The first coset is the left coset of H containing 0, the second is the left coset of H containing 1, and so forth. Observe that $gH = H$ if $g \in H$, and that $2Z \cup Z_{odd} = Z$; that is, the disjoint cosets of H partition the group $\langle Z,+ \rangle$ into the disjoint cells 2Z and Z_{odd}. Hence, the index of $\langle 2Z,+ \rangle$ is 2. Since $\langle Z,+ \rangle$ is abelian, the right cosets of H are the same as the left cosets ($gH = Hg$, $\forall\, g \in \langle Z,+ \rangle$). The cosets for $\langle Z_4,+ \rangle$ are found in a similar fashion.

The symmetric group S_n may also be partitioned into cosets. For example, $H = \{ (\,), (1,2,3), (1,3,2) \} \leq S_3$. The left cosets of H are

$$(\,)H = \{ (\,)(\,), (\,)(1,2,3), (\,)(1,3,2) \} = \{ (\,), (1,2,3), (1,3,2) \} = H;$$

$$(1,2)H = \{ (1,2)(\,), (1,2)(1,2,3), (1,2)(1,3,2) \} = \{ (1,2), (1,3), (2,3) \}.$$

The right cosets of H are

$$H(\,) = H;$$

$$H(1,2) = \{ (\,)(1,2), (1,2,3)(1,2), (1,3,2)(1,2) \} = \{ (1,2), (2,3), (1,3) \}.$$

Clearly, $(\,)H \cup (1,2)H = H(\,) \cup H(1,2) = S_3$. Also, the cardinality of each coset equals the order of H (a property that always holds).

In theory, to generate all left (right) cosets of a subgroup H, we must compute gH (Hg) for each $g \in G$. However, the Theorem of Lagrange aides in determining the number of disjoint left (right) cosets needed to partition G, i.e., the index of H in G.

> Theorem of Lagrange (Theorem 2): If H is a subgroup of a finite group G, then the order of H divides the order of G (Fraleigh, 1994).

It follows that $(G:H) = {|G|}/{|H|}$, since every coset of H contains $|H|$ elements and the cosets are disjoint. Therefore, for $H = \{ (\,), (1,2,3), (1,3,2) \}$ whose order (cardinality) is 3, yields $(S_3 : H) = {|S_3|}/{|H|} = {6}/{3} = 2$, which is consistent with our calculations above.

2.4.5 Conjugation

Tabu search applied to the TSP class of problems takes a given solution and evaluates its move neighborhood. Regardless of the specific move definition, the given solution, which is represented mathematically by a permutation in S_n, is relabeled to produce a new solution or permutation. Conjugation is a group operation that allows a permutation to be relabeled while preserving its cycle structure. By cycle structure, we mean the same number of disjoint cyclic factors where the number of letters within each factor stay the same.

> Definition: Let $g,h \in \langle G,* \rangle$, then $g^h \equiv h^{-1}gh$ is said to be the **conjugate of** g **by** h.
>
> Definition: Group elements g and h are **conjugates in** G if and only if $\exists\, x \in G$ such that $g^x = x^{-1}gx = h$.

Observe that conjugation is different from the n^{th} power operation since $h \in \langle G,* \rangle$, which is not necessarily an integer.

The concept of conjugation applied to $\langle Z,+\rangle$ and $\langle Z_4,+\rangle$ is trivial since both groups are abelian. For $a,b \in \langle Z,+\rangle$ or $\langle Z_4,+\rangle$, we have $a^b = b^{-1}ab = ab^{-1}b = ae = a$. Thus, for $g,h \in G$ the conjugate of g by h is precisely g if G is abelian.

Since S_n is non-abelian, conjugation is less trivial. Consider $p,q \in S_6$ where p = (1,3,5)(2,4,6) and q = (1,6,3). In standard form we have

$$p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 1 & 2 \end{pmatrix}, q = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 1 & 4 & 5 & 3 \end{pmatrix}, \text{ and } q^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 2 & 6 & 4 & 5 & 1 \end{pmatrix}.$$

To compute p^q implies

$$p^q = q^{-1}pq = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 2 & 6 & 4 & 5 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 1 & 2 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 1 & 4 & 5 & 3 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 4 & 2 & 6 & 1 & 3 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 1 & 4 & 5 & 3 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 4 & 2 & 3 & 6 & 1 \end{pmatrix}$$

= (1,5,6)(2,4,3). (The conjugate of p by q.)

This case demonstrates that conjugation indeed preserves the cycle structure of p.

Conjugation of permutations is a very cumbersome task, especially for large permutations; however, the next theorem provides a very useful result.

> Theorem 3: Let $p,q \in S_n$, then p^q is found by replacing each letter in p with its image in q, i.e., replace x in p with (x)q. The cycle structure of p is preserved. (Herstein, 1964)

Thus, for $p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 1 & 2 \end{pmatrix}$ and $q = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 1 & 4 & 5 & 3 \end{pmatrix}$, we have

Letter in p	1	2	3	4	5	6
Image in q	6	2	1	4	5	3

Recalling p = (1,3,5)(2,4,6) and replacing each letter x in p with its image in q, i.e., letter 1 is replaced with letter 6, letter 3 is replaced with letter 1, letter 6 is replaced with letter 3, gives $p^q = (6,1,5)(2,4,3) = (1,5,6)(2,4,3) = q^{-1}pq$ when multiplied directly as shown previously.

Conjugation in S_n is very useful in defining and describing tabu search moves for the TSP class of problems. Independent of the move definition, any permutation $q \in S_n$ may be reached by a conjugation on $p \in S_n$; that is, $\exists\, r \in S_n$ such that $p^r = q$ provided p and q share the same cycle structure. For example, consider p and q above: p = (1,3,5)(2,4,6) and q = (1,6,3). The permutation p represents a 2-TSP tour structure where salesman one takes tour $1 \rightarrow 3 \rightarrow 5 \rightarrow 1$ and salesman two takes tour $2 \rightarrow 4 \rightarrow 6 \rightarrow 2$. Conjugation of p by permutation q rearranges three nodes to yield the new solution p' = (1,5,6)(2,4,3) implying salesman one takes tour $1 \rightarrow 5 \rightarrow 6 \rightarrow 1$ and salesman two takes tour $2 \rightarrow 4 \rightarrow 3 \rightarrow 2$.

2.4.6 Templates

The TSP becomes more complex when converted to the mTSP. The mTSP is more complex because of the vast increase in feasible solutions. The only feasible solutions to the n node TSP are the n-cycles in S_n, but any product of m disjoint cycles in S_n make up the feasible solutions to the m salesman mTSP. In group theory, a TSP is easily converted to a mTSP using templates.

Colletti (1999) defines a **template** as a permutation that operates on another permutation such that its number of disjoint cycles change. The foundation for the

template comes from Rotman (1984) who states that the product of a cycle and a non-disjoint transposition is a product of two disjoint cycles. That is,

$$(a_1,\ldots,a_m,b_1,\ldots,b_n)(a_1,b_1)^{-1} = (a_1,\ldots,a_m)(b_1,\ldots,b_n).$$

Also,

$$(a_1,\ldots,a_m,b_1,\ldots,b_n) = (a_1,\ldots,a_m)(b_1,\ldots,b_n)(a_1,b_1).$$

In general,

$$(a_1,\ldots,a_i,b_1,\ldots,b_j,c_1,\ldots,c_k,d_1,\ldots,d_l)(a_1,b_1,c_1,d_1)^{-1} = (a_1,\ldots,a_i)(b_1,\ldots,b_j)(c_1,\ldots,c_k)(d_1,\ldots,d_l),$$

The permutation $(a_1,b_1,c_1,d_1)^{-1}$ is a **splitting template** since it splits the large cycle into several disjoint cycles. Also,

$$(a_1,\ldots,a_i,b_1,\ldots,b_j,c_1,\ldots,c_k,d_1,\ldots,d_l) = (a_1,\ldots,a_i)(b_1,\ldots,b_j)(c_1,\ldots,c_k)(d_1,\ldots,d_l)(a_1,b_1,c_1,d_1)$$

The permutation (a_1,b_1,c_1,d_1) is a **welding template** because it combines (welds) the disjoint cycles into one large cycle. Colletti (1999) defines a **joining template** as a template that joins several disjoint cycles into fewer larger cycles.

As an example, let p = (3,5,1,7,4,2,6), the splitting template s = $(3,4)^{-1}$ splits p into two disjoint cycles, or ps = $(3,5,1,7,4,2,6)(3,4)^{-1}$ = (3,5,1,7)(4,2,6). Similarly, if s = $(1,2)^{-1}$, then ps = $(3,5,1,7,4,2,6)(1,2)^{-1}$ = (1,7,4)(2,6,3,5). Likewise (3,5,1,7)(4,2,6)(3,4) = (3,5,1,7,4,2,6) = p, hence, w = (3,4) is a welding template for q = (3,5,1,7)(4,2,6). Finally, if p = (1,3,5,6)(2,7)(4,8,9), then j = (3,8) is a joining template since pj = (1,3,5,6)(2,7)(4,8,9)*(3,8) = (1,3,5,6)(4,8,9)(3,8) *(2,7) = (3,5,6,1,8,9,4)(2,7).

2.5 Conclusion

The archival literature for the TSP class of problems, heuristic search methods, and group theoretic metaheuristics provides the foundation for this research. The literature for the TSP class of problems is vast and provides a foundation for understanding each problem's complexities. In general, the classical methods for the TSP class of problems prevent timely optimal solutions. Hence, simpler, timelier heuristic methods must be used. The TS metaheuristic, in particular, proves capable of finding good feasible solutions to the real world TSP class of problems. Although TS is very effective, the literature is narrative in format rather than mathematical. The group theoretic metaheuristics literature explains tabu search mathematically using S_n, the symmetric group on n letters. This research uses group theory to construct, study, and clarify the underlying structures associated with the TS methodology for solving the TSP.

Chapter 3. Methodology

3.1 Introduction

This chapter presents the methodology for solving the TSP via a GTTS algorithm. First, it provides a general description and pseudo code for the GTTS algorithm. This is followed by a detailed group theoretic implementation of important tabu search concepts such as move, intensification, and diversification strategies. The chapter concludes with a description of the testing and validation of the algorithm.

3.2 Algorithm Description and Pseudo Code

The GTTS algorithm is coded with a Java programming language to take advantage of Java's object-oriented (OO) nature. The OO design encourages the reuse or alteration of existing Java code. Specifically, the Java GTTS algorithm uses code developed by Harder (2000). Harder provides the general Java classes needed for generic tabu search implementations. Harder's code requires the user to provide the specific code that defines the solution, moves, neighborhoods, intensification and diversification strategies, etc. for a specific tabu search application.

Other advantages to the Java programming language include its multi-platform portability and ease of documentation through the Javadoc tool. Javadoc allows the program to be documented with embedded code comments, and it provides the documentation in HTML format for easy navigation within any web browser. Appendix A contains the Java code and documentation for the GTTS algorithm.

A vast library of group theory C code may be found in the existing "Groups, Algorithms, and Programming" (GAP) Version 4.0 software (Schönert, 1999). The GAP software is used for computational group theory and is available for public downloading. This research does not use the GAP software since the GTTS algorithm's needed group theory code is programmed in Java. However, GAP provides additional group theory code that is needed to extend the GTTS algorithm to more complex group theory operations such as generating cosets.

The general GTTS algorithm is expressed with the simple pseudo code in Figure 1 with details of the algorithm explained in Section 3.3.

1. (Initialization Step)
    a) Set $\mathbf{I}$ = total number of iterations, n = number of nodes in the TSP, $\mathbf{C}$ = distance (cost) matrix, and l = length of tabu list.
    b) Select a starting solution $p \in \mathbf{X}$.
    c) Set $p_{Best} = p$ and let $\mathbf{z}^* = z(p_{Best})$, where $z(p)$ is the objective function.
    d) Set $\mathbf{T} = \mathbf{0}_l = (0, 0, \ldots, l)$ and $\mathbf{F} = \mathbf{0}_{n-1} = (0, 0, \ldots, n - 1)$.
2. While $i \leq \mathbf{I}$
    a) Find $N(p)$. Set $R_p = \varnothing$.
    b) Evaluate all $q \in N(p)$ using $z(q)$ and $\mathbf{C}$.
    c) Select $q \notin R_p$ such that $z(q) \leq z(\mathbf{x})$ for all $\mathbf{x} \in N(p) - R_p$.
    d) If attributes(q) $\notin \mathbf{T}$, set p = q.
    e) If attributes(q) $\in \mathbf{T}$ and $z(q) < \mathbf{z}^*$, set p = q.
    f) If attributes(q) $\in \mathbf{T}$ and $z(q) \geq \mathbf{z}^*$, then $q \in R_p$ and return to 2c).
    g) If $z(p) < \mathbf{z}^*$, repeat 1c).
    h) Update $\mathbf{T}$ and $\mathbf{F}$ with attributes(p).
    i) $i \leftarrow i + 1$.
3. Output p_{Best} and $\mathbf{z}^*$.

Figure 1. The GTTS Algorithm

3.3 Tabu Search Implementation

The GTTS algorithm implements the pseudo code above using basic tabu search concepts presented by Glover (1990). This section describes the detailed implementation

of the GTTS algorithm's starting solution, move definition and solution neighborhood, tabu criteria and aspiration criteria, and its intensification and diversification strategies.

3.3.1 Initializing the Algorithm and the Starting Solution

The first step in the GTTS algorithm requires an initialization of inputs. In this research, the total number of search iterations, **I**, is a function of the problem size n where n is the number of nodes in the TSP. Specifically, $\mathbf{I}(n) = 50n$. For example, the GTTS algorithm is performed for 2500 iterations on a 50 node TSP. The function for **I** is consistent with current TSP tabu search implementations in the literature (Carlton 1995, Ryer 1999). The input for l is discussed later in the methodology.

In the general GTTS algorithm, the set **X** in step 1b) is the set of all feasible solutions. Thus,

$$\mathbf{X} = \{\mathbf{x} \mid \mathbf{x} = (a_1, a_2, a_3, \ldots, a_n),\ a_i \in A = \{0,1,2,\ldots,n-1\},\ i \in \{1,2,\ldots,n\}\ \} \subset S_n,$$

i.e., **X** is the set of all n-cycles in S_n. Section 2.4.2 defines $A = \{1,2,\ldots,n\}$; however, without loss of generality, A may be any set of n letters. Thus, $A = \{0,1,2,\ldots,n-1\}$ is chosen here as a coding convenience since a Java array of length n is indexed from 0 to n-1.

The focus of this research is on the contributions of group theory applied to tabu search concepts. Thus, the starting search solution is irrelevant. Hence, the GTTS algorithm begins the search with an n-cycle ordered from 0 to n-1. Therefore, for an n node TSP, p = (0,1,2,3,4, . . . , n - 2,n - 1) is the starting solution.

3.3.2 Move Definition and Solution Neighborhood

The tabu search move defines the solution neighborhood, $N(p)$, in step 2a) of the GTTS algorithm. The tabu search move for our algorithm is the two letter rearrangement move, which is simply a swap of any two nodes within the TSP tour. For example, consider the incumbent solution, p, to the five node TSP. Here let p = (0,1,2,3,4). If node 2 is swapped with node 4, we have q = (0,1,4,3,2).

In group theory, the swap move is equivalent to the conjugation p^m where p is the incumbent solution and m is a transposition (a 2-cycle permutation). In the above example, p = (0,1,2,3,4) and m = (2,4) implies $p^m = (0,1,2,3,4)^{(2,4)} = (0,1,4,3,2)$. Therefore, for a swap move, m = (a,b) where a and b are the two nodes being swapped. Although the swap move is often implemented within the literature (Carlton, 1995), group theory allows the swap move to be described using algebraic equations. Since the conjugation operation applies for all $m \in S_n$, it provides a great deal of freedom in defining different rearrangement moves.

The solution neighborhood, $N(p)$, for the incumbent solution, p, is all the solutions, q, within one move of p, i.e., $N(p) = \{q \mid p^m = q, m = (a,b)$ for $a, b \in A = \{ 1, 2, \ldots, n\text{-}1\}$ and $a \neq b \}$. Note that a or $b \neq 0$. Thus, the solution p = (0,1,2,3,4,5,6,7,8,9) is uniquely expressed as an n cycle beginning at node 0. This eliminates the other n - 1 representations of p from being considered in the search, i.e., p = (0,1,2,3,4,5,6,7,8,9) = (1,2,3,4,5,6,7,8,9,0) = $\cdots$ = (9,0,1,2,3,4,5,6,7,8). As in steps 2b) through 2f), the algorithm evaluates each $q \in N(p)$ and moves to the q with the best objective function value subject to the tabu and aspiration criteria.

The size of $N(\mathrm{p})$ and the evaluation of the objective function, or tourlength, for each $\mathrm{q} \in N(\mathrm{p})$ is vital to the algorithm's efficiency. The size of $N(\mathrm{p})$, $|N(\mathrm{p})|$, is determined by the number of possible moves at each iteration. For $N(\mathrm{p}) = \{\mathrm{q} \mid \mathrm{p}^{\mathrm{m}} = \mathrm{q}, \mathrm{m} = (a,b)$ for $a, b \in A = \{1, 2, \ldots, n\text{-}1\}$ and $a \neq b\}$ we have the following matrix of moves:

$$\begin{array}{c} \overbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{n-2 \text{ moves}} \\ \left.\begin{array}{lllll} m=(1,2) & (1,3) & (1,4) & \cdots & (1,n-1) \\ m=(2,1) & (2,3) & (2,4) & \cdots & (2,n-1) \\ m=(3,1) & (3,2) & (3,4) & \cdots & (3,n-1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m=(n-2,1) & (n-2,2) & (n-2,3) & \cdots & (n-2,n-1) \\ m=(n-1,1) & (n-1,2) & (n-1,3) & \cdots & (n-1,n-2) \end{array}\right\} n-1 \text{ moves} \end{array} \begin{array}{l} a=1 \\ a=2 \\ a=3 \\ \vdots \\ a=n-2 \\ a=n-1 \end{array}$$

Thus, $|N(\mathrm{p})| = (n-2)(n-1)$. However, $\mathrm{m}_1 = (a,b) = \mathrm{m}_2 = (b,a)$ implying

$$|N(\mathrm{p})| = \frac{(n-2)(n-1)}{2}.$$

Colletti (1999) presents an efficient way to evaluate the objective function for each $\mathrm{q} \in N(\mathrm{p})$. The objective function defined in step 1c) of the GTTS algorithm is simply the tourlength of p, $T(\mathrm{p})$. If $T(\mathrm{p})$ is the tourlength for $\mathrm{p} \in S_n$, then

$$T(\mathrm{p}) = \sum_{i=1}^{n} c_{i,(i)\mathrm{p}} = \sum_{i \in \mathrm{mov(p)}} c_{i,(i)\mathrm{p}}$$

where $c_{i,(i)\mathrm{p}}$ is the $i,\mathrm{p}(i)^{\text{th}}$ entry of the n by n zero-diagonal cost matrix $\mathbf{C}$, and mov(p) is the set of all letters moved by p, that is, the set of all letters within the permutation p. For example, if $\mathrm{p} = (1,3,4,2,7) \in S_7$, then $\mathrm{mov(p)} = \{1, 2, 3, 4, 7\}$ and $T(\mathrm{p}) =$

$$c_{1,(1)\mathrm{p}} + c_{2,(2)\mathrm{p}} + c_{3,(3)\mathrm{p}} + c_{4,(4)\mathrm{p}} + c_{7,(7)\mathrm{p}} = c_{1,3} + c_{2,7} + c_{3,4} + c_{4,2} + c_{7,1}.$$

If $T(p) \neq T(q)$ for $p,q \in S_n$, then there exists noncommon arcs between them whose tails are the set $\text{mov}(qp^{-1})$. If p and q both share arc i-j, then $(i)p = (i)q = j$, but $(j)p^{-1} = i$ implying $(i)qp^{-1} = i$. The difference in tourlengths, denoted $\Delta(p,q)$, is:

$$\Delta(p,q) = T(p) - T(q) = \sum_{i \in \text{mov}(qp^{-1}))} [c_{i,(i)p} - c_{i,(i)q}].$$

Hence, for $q \in N(p)$, $T(q) = T(p) - \Delta(p,q)$. For large n, if $T(p)$ is known and $\text{mov}(qp^{-1})$ is a small set, then computing $\Delta(p,q)$ is a simple calculation. Further, calculating $T(q)$ using $\Delta(p,q)$ is much easier than calculating $T(q)$ directly. However, to calculate $\Delta(p,q)$, we must know $(i)q$, but $(i)q = (i)p^m = (i)m^{-1}pm = (((i)m^{-1})p)m$. Hence, only p and m are required to find $T(q)$, that is, no knowledge of q is required.

Our tabu search move $p^m = q$, where $m = (a,b)$ produces the set $\text{mov}(qp^{-1})$ where $|\text{mov}(qp^{-1})| \leq 4$ for all $m = (a,b)$, since $p^m = q$ implies $qp^{-1} = p^m p^{-1} = (m^{-1}pm)p^{-1} = m^{-1}(pmp^{-1}) = m^{-1}m^{p^{-1}}$. Here $m^{-1} = m$, since m is a transposition, and $m^{p^{-1}}$ is also a transposition since conjugation preserves cycle structure. Therefore, the product qp^{-1} involves at most four letters implying $|\text{mov}(qp^{-1})| \leq 4$. Specifically, $qp^{-1} = m^{-1}m^{p^{-1}} = (a,b)((a)p^{-1},(b)p^{-1})$ implying $\text{mov}(qp^{-1}) = \{a, b, (a)p^{-1}, (b)p^{-1}\}$. Observe that $(a)p^{-1}$ is the letter preceding a in p and $(b)p^{-1}$ is the letter preceding b in p. If a and b are adjacent nodes, then $(b)p^{-1} = a$ and $|\text{mov}(qp^{-1})| = 3$, otherwise $|\text{mov}(qp^{-1})| = 4$. For example, consider p = (0,1,3,4,2,5) and m = (3,5), then q = (0,1,5,4,2,3) and $\text{mov}(qp^{-1}) = \{a, b, (a)p^{-1}, (b)p^{-1}\} = \{3, 5, (3)p^{-1}, (5)p^{-1}\} = \{3, 5, 1, 2\}$ since 1 precedes 3 and 2 precedes 5 in p.

To illustrate tourlength calculations, consider the 10 Ohio city TSP with current solution p = (0,1,2,3,4,5,6,7,8,9) and cost matrix **C**. The actual cost matrix is:

		0	1	2	3	4	5	6	7	8	9
	0 Akron	0	24	227	33	121	186	150	161	122	46
	1 Canton	24	0	224	56	118	184	150	159	140	51
	2 Cincinnati	227	224	0	239	106	52	128	71	198	273
	3 Cleveland	33	56	239	0	141	197	150	172	107	66
C =	4 Columbus	121	118	106	141	0	68	90	43	128	167
	5 Dayton	186	184	52	197	68	0	76	25	152	232
	6 Lima	150	150	128	150	90	76	0	66	81	196
	7 Springfld	161	159	71	172	43	25	66	0	127	207
	8 Toledo	122	140	198	107	128	152	81	127	0	168
	9 Youngstown	46	51	273	66	167	232	196	207	168	0

This yields $T(\mathrm{p}) = \sum_{i=0}^{9} c_{i,(i)\mathrm{p}} = c_{0,1} + c_{1,2} + c_{2,3} + c_{3,4} + \cdots + c_{9,0} = 24 + 224 + 239 + 141 + 68$ + 76 + 66 + 127 + 168 + 46 = 1179. Let m = (4,9), then q = (0,1,2,3,9,5,6,7,8,4) and $\mathrm{mov}(\mathrm{qp}^{-1}) = \{4, 9, 3, 8\}$. This gives $\Delta(\mathrm{p},\mathrm{q}) = \sum_{i \in \mathrm{mov}(\mathrm{qp}^{-1}))} [c_{i,(i)\mathrm{p}} - c_{i,(i)\mathrm{q}}] = \sum_{i \in \{3,4,8,9\}} [c_{i,(i)\mathrm{p}} - c_{i,(i)\mathrm{q}}]$

$= [c_{3,(3)\mathrm{p}} - c_{3,(3)\mathrm{q}}] + [c_{4,(4)\mathrm{p}} - c_{4,(4)\mathrm{q}}] + [c_{8,(8)\mathrm{p}} - c_{8,(8)\mathrm{q}}] + [c_{9,(9)\mathrm{p}} - c_{9,(9)\mathrm{q}}] =$

$[c_{3,4} - c_{3,9}] + [c_{4,5} - c_{4,0}] + [c_{8,9} - c_{8,4}] + [c_{9,0} - c_{9,5}]$ = [141 - 66] + [68 - 121] + [168 - 128] + [46 - 232] = 75 + -53 + 40 + -186 = -124. Thus, $T(\mathrm{q}) = T(\mathrm{p}) - \Delta(\mathrm{p},\mathrm{q}) = 1179 +$ 124 = 1303. Computing $T(\mathrm{q})$ directly gives $T(\mathrm{q}) = \sum_{i=0}^{9} c_{i,(i)\mathrm{q}} = c_{0,1} + c_{1,2} + c_{2,3} + c_{3,9} +$

$c_{4,0} + c_{5,6} + c_{6,7} + c_{7,8} + c_{8,4} + c_{9,5} = 24 + 224 + 239 + 66 + 121 + 76 + 66 + 127 + 128 +$ 232 = 1303.

3.3.3 Conjugation and *k*-opt Moves

There is a relationship between *m*-letter rearrangement moves by conjugation and the classical *k*-opt moves found in the literature. The GTTS algorithm move is the two-letter rearrangement move, which is simply a swap of any two nodes within the TSP tour. Recall that the *k*-opt move is one where *k* arcs are dropped from the existing solution and then *k* new arcs are added back for an improving solution. The 2-letter rearrangement move, or swap, is a type of *k*-opt move. Specifically, the swap move $p^m = q$ where m = (*a*,*b*) is a 4-opt move if *a* and *b* are non-adjacent nodes in p and a 3-opt move if *a* and *b* are adjacent nodes. For example, consider the incumbent solution, p, to the 5 node TSP. Here let p = (0,1,2,3,4). If node 2 is swapped with node 4, we have q = (0,1,4,3,2). For this move, arcs 1-2, 2-3, 3-4, and 4-0 are cut while arcs 1-4, 4-3, 3-2, and 2-0 are added. We assume here that arc *i*-*j* is not the same as arc *j*-*i*.

In group theory, *m*-letter rearrangement moves by conjugation are *k*-opt moves. The literature rarely demonstrates *k*-opt moves for $k > 4$ because of their complexity. However, conjugation performs *k*-opt moves in a simple and straightforward manner. For example, suppose p = (0,2,3,1,5,6,4) and m = (1,6,2). The permutation m is a 3 - cycle and equates to a three-letter rearrangement move. Thus, $p^m = q = (0,1,3,6,5,2,4)$, and m = (1,6,2) is a 6-opt move since arcs 0-2, 2-3, 3-1,1-5, 5-6, 6-4 are cut in p and arcs 0-1, 1-3, 3-6, 6-5, 5-2, 2-4 are added back in q. In general, an *m*-letter rearrangement move by conjugation is a 2*m*-opt move if the letters being moved are all non-adjacent in the TSP tour.

3.3.4 Tabu Criteria and Aspiration Criteria

A neighboring solution q $\in$ N(p) is evaluated by its tourlength, T(q), subject to tabu and aspiration criteria. The algorithm uses recency (short-term memory) to determine if the new solution q has been visited in recent search iterations. The GTTS algorithm uses a "tabu list", **T**, of length $l = 2t$ to determine if q is "tabu", where t is the tabu tenure. The value of t is a function of n, the number of nodes in the TSP. The algorithm uses different values of t to demonstrate how the solution's quality depends on the tabu criteria. Here, t takes on two values for each problem instance. Namely, $t = \lfloor 0.10\alpha n \rfloor$ for $\alpha = 1$ and 2, where n is the number of nodes in the TSP. At each iteration i, the GTTS algorithm records the move attributes for m = (a,b) that yields p^m = q (p is the solution for iteration (i -1) and q is the solution for iteration i) in **T** as stated in step 2h) of the algorithm. Thus, attributes(q) for **T** are the nodes swapped, or moved, by m. Specifically, node a is placed on line ($2t$ - 1) and node b is placed on line $2t$ (final line) of the tabu list **T**. The move for iteration i -1, which was previously stored on lines ($2t$ - 1) and $2t$, is moved to lines ($2t$ - 3) and ($2t$ - 2) of **T**, while the move for iteration (i - 2) is moved from lines ($2t$ - 3) and ($2t$ - 2) to lines ($2t$ - 5) and ($2t$ - 4), and so forth until the move for iteration i - (t-1) is moved from lines 3 and 4 to lines 1 and 2. The move for iteration i - t is dropped from the list. Hence, the tabu list **T** remembers the past t moves as required. At each iteration, any m = (a,b) is classified tabu if a or b is in the tabu list array. For example, suppose t = 1. Further, suppose the move m = (2,4) for p = (0,1,2,3,4) yielding q = (0,1,4,3,2) gives the best objective function value. Then, q is the new incumbent solution and m = (2,4) is added to the tabu list, i.e., **T** = (2, 4). For the

next t iterations, any m = (a,b) where $a \in \{2, 4\}$ or $b \in \{2, 4\}$ is tabu. Each solution q = p^m, where m contains tabu letters, is still evaluated in step 2b). If q is selected as the best neighbor, but $T(q) \geq T(p_{Best})$ then q is placed in the set of restricted moves R_p as in step 2f).

F is the long-term memory for the GTTS algorithm. The list **F** is of length n - 1 and records the number of times, or frequency, that a letter is swapped. Hence, for the move m = (2,4), the second and fourth lines of **F** are incremented by one in step 2h) as a record that nodes 2 and 4 have been swapped.

It is important that the tabu criterion keeps the search from returning to past solutions. Theorem 3 in Section 2.4.5 and a solution's unique representation ensure that the solution p is not revisited within t iterations. Essentially, Theorem 3 states that for some m = (x,y), nodes a and b are moved in p, if and only if $x \in \{a, b\}$ or $y \in \{a, b\}$. The unique representation of p ensures that j^{th} position of the cycle uniquely determines the cycle. For example, if p = (0,1,2,3,4) = (1,2,3,4,0) = (2,3,4,0,1) then a = 1 may be in any cycle position of p and still equate to the same cyclic permutation. However, if p is uniquely represented by p = (0,1,2,3,4), then a = 1 in p is only allowed in the second position.

At iteration i, any m = (a,b) may be taken if p^m = q produces the best known objective function value encountered in the first i iterations. This is the algorithm's aspiration criteria that is used to counterbalance the effect of the tabu list. This equates to step 2e) in the GTTS algorithm. As an illustration, let m = (2,4) be tabu. The move m = (2,3) for p = (0,1,4,3,2) giving q = (0,1,4,2,3) is permitted if q produces the best objective function value to date, i.e., $z(q) < \mathbf{z}^*$ or $T(q) < T(p_{Best})$.

Summarizing the previous sections, the GTTS algorithm is stated again in full detail.

1. (Initialization Step)
   a) Set $\mathbf{I} = 50n - \mathbf{E}$, n = number of nodes in the TSP, $\mathbf{C}$ = distance (cost) matrix, $l = 2t$ where $t = \lfloor 0.10\alpha n \rfloor$ for $\alpha = 1, 2$, and $\mathbf{E}$ is defined in the intensification strategy.
   b) Set the starting solution $p = (0,1,2,3, \ldots, n\text{-}1) \in S_n$.
   c) Set $p_{Best} = p$ and let $\mathbf{z}^* = T(p_{Best})$, where $T(p)$ is the objective function.
   d) Set $\mathbf{T} = \mathbf{0}_l = (0, 0, \ldots, l)$ and $\mathbf{F} = \mathbf{0}_{n-1} = (0, 0, \ldots, n - 1)$.
2. While $i \leq \mathbf{I}$
   a) Find $N(p) = \{q \mid p^m = q, m = (a,b)$ for $a, b \in A = \{ 1, 2, \ldots, n\text{-}1\}$ and $a \neq b \}$. Set $R_p = \varnothing$.
   b) Evaluate all $q \in N(p)$ using $T(q) = T(p) - \Delta(p,q)$ and $\mathbf{C}$.
   c) Select $q \notin R_p$ such that $T(q) \leq T(\mathbf{x})$ for all $\mathbf{x} \in N(p) - R_p$.
   d) If $m_q = (a,b)$ and a or $b \notin \mathbf{T}$, set $p = q$. Here, m_q is the move from p to the q selected in 2c)
   e) If a or $b \in \mathbf{T}$ and $T(q) < T(p_{Best})$, set $p = q$.
   f) If a or $b \in \mathbf{T}$ and $T(q) \geq T(p_{Best})$, then $q \in R_p$ and return to 2c).
   g) If $T(p) < T(p_{Best})$, repeat 1c).
   h) Update $\mathbf{T}$ by resorting $\mathbf{T}$ and adding a to line $(2t - 1)$ and b to line $2t$. Update $\mathbf{F}$ by incrementing the a^{th} and b^{th} lines by one.
   i) $i \leftarrow i + 1$.
3. Output p_{Best} and $T(p_{Best})$.

Figure 2. The GTTS Refined Algorithm

3.3.5 Intensification Strategy

The GTTS algorithm employs an elite list intensification strategy (Glover and Laguna, 1997). This strategy intensifies the search around the best r solutions found within the first $50n - \mathbf{E}$ iterations of the search. In the GTTS algorithm, $r = 5$ and $\mathbf{E} = (0.40)\mathbf{I}_{\mathbf{Total}} = (0.40)50n = 20n$ is the total number of intensification iterations. Clearly, the algorithm requires a memory of its elite solutions. Let $\mathbf{S} = (p_1, p_2, p_3, p_4, p_5)$ be the list of elite solutions for the first $50n - \mathbf{E} = 30n$ iterations, where $p_1 = p_{Best}$, p_2 is the next

best and so forth for p_3 through p_5. Then step 1c) of the GTTS algorithm above should read

c) In **S**, set $p_j = p_{j-1}$ for $j = 5, 4, 3, 2$, and set $p_{Best} = p_1 = p$ and let $\mathbf{z}^* = T(p_{Best})$, where $T(p)$ is the objective function.

The intensification search follows the same GTTS algorithm defined above with a few modifications. The GTTS intensification algorithm is:

1. (Initialization Step)
   a) Set $\mathbf{I}_{\mathbf{Intensify}} = \mathbf{E} = 20n$, n = number of nodes in the TSP, **C** = distance (cost) matrix, $l = 3t$ where $t = \lfloor 0.10\alpha n \rfloor$ for $\alpha = 1, 2$, and **S** = $(p_1, p_2, p_3, p_4, p_5)$ is the list of elite solutions found in the first $30n$ iterations.
   b) Set the starting solution $p_j \in S_n$ for $j = 1, 2, 3, 4, 5$.
   c) Set $p_{Best} = p$ and let $\mathbf{z}^* = T(p_{Best})$, where $T(p)$ is the objective function.
   d) Set $\mathbf{T} = \mathbf{0}_l = (0, 0, \ldots, l)$ and $\mathbf{F} = \mathbf{0}_{n-1} = (0, 0, \ldots, n-1)$.
2. While $i \leq \mathbf{I}_{\mathbf{Itensify}}$
   a) Find $N(p) = \{q \mid p^m = q$, for selected $m = (a,b,c)$ for $a, b \in A = \{1, 2, \ldots, n-1\}$ and $a \neq b \neq c\}$. Set $R_p = \varnothing$.
   b) Evaluate all $q \in N(p)$ using $T(q) = T(p) - \Delta(p,q)$ and **C**.
   c) Select $q \notin R_p$ such that $T(q) \leq T(\mathbf{x})$ for all $\mathbf{x} \in N(p) - R_p$.
   d) If $m_q = (a,b,c)$ and a, b, or $c \notin \mathbf{T}$, set $p = q$. Here, m_q is the move from p to the q selected in 2c)
   e) If a, b, or $c \in \mathbf{T}$ and $T(q) < T(p_{Best})$, set $p = q$.
   f) If a, b, or $c \in \mathbf{T}$ and $T(q) \geq T(p_{Best})$, then $q \in R_p$ and return to 2c).
   g) If $T(p) < T(p_{Best})$, repeat 1c).
   h) Update **T** by resorting **T** and adding a to line $(3t - 2)$, b to line $(3t - 1)$, and c to line $3t$. Update **F** by incrementing the a^{th}, b^{th}, and c^{th} lines by one.
   i) $i \leftarrow i + 1$.
3. Output p_{Best} and $T(p_{Best})$.

Figure 3. The GTTS Intensification Algorithm

First, observe that $\mathbf{I}_{\mathbf{Intensify}} = \mathbf{E}$. Thus, if $\mathbf{I}_{\mathbf{Total}} = 50n$ and $n = 50$, then **E** = (0.40)2500 = 1000 iterations, and each j^{th} elite solution is intensified for $\mathbf{I}_{\mathbf{Intensify}} = 1000$ iterations. Here, the j^{th} elite solution acts as the starting solution p in step 1b).

Second, the tabu list l is now of length $3t$, but is modified in the same manner as defined previously. The change in l is due to a different move definition. Observe in step 2a) that $N(\mathrm{p}) = \{\mathrm{q} \mid \mathrm{p}^{\mathrm{m}} = \mathrm{q}$, for selected $\mathrm{m} = (a,b,c)$ for $a, b \in A = \{1, 2, \ldots, n\text{-}1\}$ and $a \neq b \neq c\}$. Here, the neighborhood of p, $N(\mathrm{p})$, only includes a selected number of 3-cycles to conjugate with p. Therefore, our intensification move is a three-letter rearrangement move.

$N(\mathrm{p})$ only includes a selected number of 3-cycles because the size of $N(\mathrm{p})$, $|N(\mathrm{p})|$, becomes too large if all 3-cycles are considered. To illustrate, consider the solution neighborhood $N(\mathrm{p}) = \{\mathrm{q} \mid \mathrm{p}^{\mathrm{m}} = \mathrm{q}$ for all $\mathrm{m} = (a,b,c) = ((i)\mathrm{p}, (i+j+1)\mathrm{p}, (i+j+k+2)\mathrm{p})$ for $i = 0, \ldots, n\text{-}4$, $j = 0, \ldots, ((n\text{-}4) - i)$ for each i, and $k = 0, \ldots, ((n\text{-}4)-(i+j))$ for each i and $j\}$. Without loss of generality, let $\mathrm{p} = (0,1,2,3,\ldots,n\text{-}1)$. $N(\mathrm{p})$ is constructed with the looping structure in Figure 4.

```
For i = 0, . . . , n-4
        a = (i)p
        For j = 0, . . . , ((n-4) - i)
                b = (i+j+1)p
                For k = 0, . . . , ((n-4) - (i+j))
                        c = (i+j+k+2)p
```

Figure 4. Looping Structure for $N(\mathrm{p})$

As a simple illustration, let $i = 1$, $j = 2$, and $k = 4$, then $\mathrm{m} = (a,b,c) = ((1)\mathrm{p},(4)\mathrm{p},(9)\mathrm{p}) = (2,5,10)$ when $\mathrm{p} = (0,1,2,3,\ldots,n\text{-}1)$. Thus, when $i = 0$, we have the following group of moves:

$j\downarrow$	$k\rightarrow$						
0	(1,2,3)	(1,2,4)	(1,2,5)	(1,2,6)	$\cdots$	(1,2,$n-2$)	(1,2,$n-1$)
1	(1,3,4)	(1,3,5)	(1,3,6)	$\cdots$	(1,3,$n-2$)	(1,3,$n-1$)	
2	(1,4,5)	(1,4,6)	$\cdots$	(1,4,$n-2$)	(1,4,$n-1$)		
$\vdots$	$\vdots$						
($n-4$)	(1,$n-2$,$n-1$)						

Similarly, i = 1 gives the moves

$j\downarrow$	$k\rightarrow$						
0	(2,3,4)	(2,3,5)	(2,3,6)	(2,3,7)	$\cdots$	(2,3,$n-2$)	(2,3,$n-1$)
1	(2,4,5)	(2,4,6)	(2,4,7)	$\cdots$	(2,4,$n-2$)	(2,4,$n-1$)	
2	(2,5,6)	(2,5,7)	$\cdots$	(2,5,$n-2$)	(2,5,$n-1$)		
$\vdots$	$\vdots$						
($n-3$)	(2,$n-2$,$n-1$)						

A similar group of moves exists for each i. Observe that there are $\sum_{x=1}^{n-3} x = \frac{(n-3)(n-2)}{2}$ moves for $i = 0$ and $\sum_{x=1}^{n-4} x = \frac{(n-4)(n-3)}{2}$ moves for $i = 1$. For example, if $n = 8$ then $i =$ 0 produces the following moves.

$j\downarrow$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	
0	(1,2,3)	(1,2,4)	(1,2,5)	(1,2,6)	(1,2,7)	$x=5$
1	(1,3,4)	(1,3,5)	(1,3,6)	(1,3,7)		$x=4$
2	(1,4,5)	(1,4,6)	(1,4,7)			$x=3$
3	(1,5,6)	(1,5,7)				$x=2$
4	(1,6,7)					$x=1$

Thus, $\sum_{x=1}^{n-3} x = \frac{(n-3)(n-2)}{2} = \sum_{x=1}^{5} x = \frac{(5)(6)}{2} = 15$ total moves for $i = 0$. In general, we have $\frac{((n-3)-i)((n-2)-i)}{2}$ moves for each i = 0, 1, 2, . . . , n - 4. Hence,

$$|N(\mathrm{p})| = \sum_{i=0}^{n-4} \frac{((n-3)-i)((n-2)-i)}{2}.$$

Clearly, $N(\mathrm{p})$ becomes extremely large as n grows. This particular neighborhood structure still does not include all the possible three-letter rearrangement moves on p. For example, m = (1,3,2) $\notin$ $N(\mathrm{p})$ for p = (0,1,2, . . . ,n-1).

The GTTS intensification algorithm explores two different neighborhood structures to demonstrate their effect on the solution quality. The first neighborhood structure is a scaled version of the neighborhood described above. Here, $N_1(\mathrm{p}) = \{\mathrm{q} \mid \mathrm{p}^{\mathrm{m}} = \mathrm{q}$ for all m = (a,b,c) = ((i)p, (i+j+1)p, (i+j+k+2)p) for i = 0, 1, 2, 3; j = 0, . . . , ((n-4) - i) for each i; and k = 0, . . ., ((n-4)-(i+j)) for each i and j }. The scaling is due to the limitation on i. Hence, the first neighborhood structure $N_1(\mathrm{p})$ is generated by the looping structure below.

```
For i = 0, 1, 2, 3
        a = (i)p
        For j = 0, . . . , ((n-4) - i)
                b = (i+j+1)p
                For k = 0, . . . , ((n-4) - (i+j))
                        c = (i+j+k+2)p
```

Figure 5. Looping Structure for $N_1(\mathrm{p})$

The size of $N_1(\mathrm{p})$ is $|N_1(\mathrm{p})| = \sum_{i=0}^{3} \frac{((n-3)-i)((n-2)-i)}{2}$. The algebra simplifies to $|N_1(\mathrm{p})| = (n-3)^2 + (n-5)^2$. This neighborhood structure limits the effectiveness of the tabu list since any letter moved into (i)p for i = 0, 1, 2, 3 by m may remain tabu for at most three iterations.

The second neighborhood structure used by the GTTS intensification algorithm is given by $N_2(\mathrm{p}) = \{\mathrm{q} \mid \mathrm{p}^{\mathrm{m}} = \mathrm{q}$ for all m = (a,b,c) = ((i)p, (i+j+1)p, (i+j+2)p) for i = 0, 1, 2, .

. . , n-4, and $j = 0, \ldots, ((n\text{-}4) - i)$ for each i}. Thus, N_2(p) is generated by the looping structure below.

For $i = 0, 1, \ldots, n$-4
  $a = (i)$p
  For $j = 0, \ldots, ((n\text{-}4) - i)$
    $b = (i+j+1)$p
    $c = (i+j+2)$p

Figure 6. Looping Structure for N_2(p)

The size of N_2(p), $|N_2(\mathrm{p})|$, is much smaller than that of N_1(p). Applying the looping structure for N_2(p) to p = (0,1,2, . . .,n-1) gives the following 3-cycle moves:

$i\downarrow$	$j\rightarrow$					
0	(1,2,3)	(1,3,4)	(1,4,5)	(1,5,6)	$\cdots$	$(1,n-2,n-1)$
1	(2,3,4)	(2,4,5)	(2,5,6)	$\cdots$	$(2,n-2,n-1)$	
2	(3,4,5)	(3,5,6)	$\cdots$	$(3,n-2,n-1)$		
$\vdots$	$\vdots$					
$(n-4)$	$(n-3,n-2,n-1)$					

Thus, $|N_2(\mathrm{p})| = \sum_{x=1}^{n-3} x = \frac{(n-3)(n-2)}{2}$. In conclusion, the GTTS intensification algorithm follows the same process as the general GTTS algorithm. However, the details of the search, such as the number of iterations, move definition, tabu list, and initial starting solution, differ.

3.3.6 Diversification Strategy

The GTTS algorithm uses frequency (long-term memory) to diversify the search into unexplored regions of the solution space. The diversification strategy uses the ideas of influence and quality (Glover and Laguna, 1997). Any tabu search move has both a measure of influence and a measure of quality. In our case, the measure of move

influence is the order of the move, m, where m = $(a_1, a_2, a_3, \ldots, a_m)$, and the measure of move quality is the objective function value, or tourlength, $T(q)$ for $q = p^m$. Recall that m = $(a_1, a_2, a_3, \ldots, a_m)$, an m-cycle, has order m. In the general GTTS algorithm, the influence for any move is 2, since we define our move as m = (a,b), and in the GTTS intensification algorithm, the influence for any move is 3 since m = (a,b,c).

When the search experiences little change in quality, the diversification strategy is employed. In the GTTS algorithm, this occurs after $\mathbf{U} = \max\{5, (0.005)\mathbf{I}\}$ consecutive non-improving moves have been chosen. Specifically, the GTTS algorithm diversifies the search by making highly influential moves; that is, the search diversifies by making a rearrangement move of order m. In the GTTS algorithm, m is a function of the problem size n. Namely, $m = \max\{5, (0.15)n\}$. Hence, we diversify if at least 0.5 percent of the total number of iterations are consecutively non-improving by rearranging at least 15 percent of the nodes within the TSP tour.

Recall that **F** is the long-term memory for the GTTS algorithm. The list **F** is of length n - 1 and records the number of times that a letter is moved. The GTTS algorithm uses **F** to determine the nodes that need to be moved and the nodes that need to remain stable. Specifically, the m least moved nodes are moved by $m_D = (a_1, a_2, a_3, \ldots, a_m)$, where a_k is the k^{th} least moved node.

As an example, suppose p = (0,2,3,1,7,8,9,5,4,10,11,6,12) is the fifth non-improving move made and suppose nodes 11, 3, 10, 8, 12 are found, after searching **F**, to be the five least moved nodes. Then, $p_{new} = p^{m_D}$, where m_D = (11,3,10,8,12), implying p_{new} = (0,2,10,1,7,12,9,5,4,8,3,6,11). Thus, p_{new} becomes the new incumbent solution,

i.e. $p_{new} = p$ in line 1b) of the GTTS algorithm. Thus, the GTTS algorithm diversifies by jumbling the incumbent solution and restarting the search. This jumbling is accomplished by an *m*-letter rearrangement of the nodes that are rarely moved.

3.4 Testing and Validation

The Java GTTS algorithm is tested and validated on several TSP data sets. These data sets range from 10 to 76 nodes and include both symmetric and asymmetric instances. The smallest problem, the Ohio10 problem, is taken from Moore (1999). The remaining data sets are taken from TSPLIB, which is a library of TSP instances from various sources and is compiled by Reinelt (2000). Computational results are compared to the optimal solutions given by Reinelt (2000). The results are presented in Chapter 4. The actual problem instances used for testing are given in the table below.

Table 1. TSP Data Sets

Problem Name	Description	Number of Nodes	Type
Ohio10	10 Ohio cities	10	Symmetric
Gr17	17-city problem	17	Symmetric
Gr21	21-city problem	21	Symmetric
Gr24	24-city problem	24	Symmetric
Fri26	26-city problem	26	Symmetric
Swiss42	42-city problem	42	Symmetric
Att48	48 CONUS state capitals	48	Symmetric
Gr48	48-city problem	48	Symmetric
Eil51	51-city problem	51	Symmetric
Berlin52	52 locations in Berlin	52	Symmetric
St70	70-city problem	70	Symmetric
Eil76	76-city problem	76	Symmetric
Br17	None	17	Asymmetric
Ftv35	None	36	Asymmetric
P43	None	43	Asymmetric

3.5 Conclusion

A specific instance of the GTTS algorithm is coded in the Java programming language using the generic tabu search framework developed by Harder (2000). Appendix A contains the specific GTTS Java code as a reference. The GTTS algorithm employs a 2-letter rearrangement move, or swap, by conjugating the incumbent tour p with a transposition m. The GTTS intensification algorithm also uses conjugation to perform 3-letter rearrangement moves. The diversification strategy for GTTS restarts the search with a new starting solution by making an m-letter rearrangement of the incumbent tour using the m most unmoved nodes. The results of the GTTS algorithm follow in Chapter 4.

Chapter 4. Results

4.1 Introduction

An instance of the GTTS algorithm is coded in the Java programming language. The Java program is a very simple implementation of GTTS, and lays no claim as the ultimate GTTS implementation. In fact, there are several ways to improve this Java implementation in efficiency and sophistication, some of which are mentioned in Chapter 5. Thus, the purpose of the Java GTTS implementation is to demonstrate the GTTS methodology described in Chapter 3 and provide an indication of its effectiveness. Specifically, this chapter provides computational results and analytical conclusions for the Java GTTS algorithm.

4.2 Algorithm Results

The objective of the GTTS algorithm is to find the minimal tourlength, $T(\mathrm{p})$, for a given TSP instance within a set number of iterations **I**. Thus, a natural measure of effectiveness for the Java GTTS code is its ability to find a TSP tour, p, that minimizes, or nearly minimizes, the tourlength $T(\mathrm{p})$. In addition to finding good solutions, an important measure for any heuristic is the amount of time it takes to find the solution p. Hence, the execution time is provided, although timeliness depends greatly on the efficiency of the actual code.

The value r is used as a measure of solution quality, where

$$r = \frac{\text{Minimal Tourlength Found by GTTS}}{\text{Optimal Tourlength}} = \frac{T(\mathrm{p}_{\mathrm{Best}})}{T(\mathrm{p}_{\mathrm{Optimal}})}.$$

Clearly, $r = 1$ indicates that $T(p_{Best}) = T(p_{Optimal})$ implying that $r \approx 1$ is desired. Several TSP heuristics provide a worst case value for r as a measure of solution quality. For example, the nearest neighbor heuristic referenced in Section 2.3 has worst-case $r \leq \frac{1}{2}(\log_2 n) + \frac{1}{2}$ where n is the problem size (Bodin *et al.*, 1993).

The execution time is given in seconds, and is defined as amount of time required to complete **I** iterations using a Pentium II 350 MHz/64 MB RAM machine. Table 2 presents the optimal solution, $T(p_{Optimal})$, best found GTTS solution, $T(p_{Best})$, r, and average execution time for each test problem. The execution time is shown for intensification neighborhood one, $N_1(p)$, and intensification neighborhood two, $N_2(p)$ as defined in Section 3.3.5. Also, each problem iterates through $\mathbf{I} = 30n + 5(20n)$ total iterations.

Table 2. GTTS Best Results

Problem	$T(p_{Optimal})$	$T(p_{Best})$	r	Execution Time (sec.) ($N_1(p)$)	Execution Time (sec.) ($N_2(p)$)
Ohio10	678	678	1	3.13	2.61
gr17	2085	2181	1.046043	7.07	7.51
gr21	2707	2858	1.055781	42.64	14.90
gr24	1272	1338	1.051887	76.61	25.28
fri26	937	941	1.004269	110.67	30.39
swiss42	1273	1427	1.120974	816.52	201.15
Att48	10628	11239	1.05749	1543.68	364.75
gr48	5056	5710	1.129351	1549.49	372.32
eil51	426	522	1.225352	2340.27	433.75
Berlin52	7542	8649	1.146778	2768.87	799.25
st70	675	895	1.325926	11,902.25	1880.50
eil76	538	675	1.254647	17,856.00	2378.08
br17	39	39	1	18.80	7.22
Ftv35	1473	1620	1.099796	388.81	110.73
p43	5620	5652	1.005694	1005.65	225.43
		Averages	1.101599	2695.36	456.92

In GTTS, intensification and diversification should improve the solution quality. Thus, it is important to see if the solution quality significantly improves as the diversification and intensification strategies are implemented. Table 3 through Table 6 show the best found solutions for the general GTTS search, i.e., GTTS without intensification or diversification, then GTTS with diversification, then GTTS with intensification, and finally, GTTS with intensification and diversification.

Table 3. GTTS Results without Intensification or Diversification

Problem	$T(p_{Optimal})$	$T(p_{Best})$	r	General Iterations
Ohio10	678	678	1	300
gr17	2085	2270	1.088729	510
gr21	2707	3223	1.190617	630
gr24	1272	1383	1.087264	720
fri26	937	957	1.021345	780
swiss42	1273	1808	1.420267	1260
Att48	10628	17,589	1.654968	1440
gr48	5056	6091	1.204707	1440
eil51	426	638	1.497653	1530
Berlin52	7542	10,138	1.344206	1560
st70	675	1017	1.506667	2100
eil76	538	771	1.433086	2280
br17	39	41	1.051282	510
Ftv35	1473	1890	1.283096	1080
p43	5620	5668	1.008541	1290
		Averages	1.252828	1162

Table 4. GTTS Results with Diversification

Problem	$T(p_{Optimal})$	$T(p_{Best})$	r	General Iterations
Ohio10	678	685	1.010324	300
gr17	2085	2220	1.064748	510
gr21	2707	2872	1.060953	630
gr24	1272	1399	1.099843	720
fri26	937	957	1.021345	780
Swiss42	1273	1564	1.228594	1260
Att48	10628	13,296	1.251035	1440
gr48	5056	6217	1.229628	1440
eil51	426	542	1.2723	1530
Berlin52	7542	8,813	1.168523	1560
st70	675	962	1.425185	2100
eil76	538	707	1.314126	2280
br17	39	41	1.051282	510
Ftv35	1473	1890	1.283096	1080
p43	5620	5700	1.014235	1290
		Averages	1.166348	1162

Table 5. GTTS Results with Intensification

Problem	$T(p_{Optimal})$	$T(p_{Best})$	r	General Iterations	Intensify Iterations
Ohio10	678	678	1	300	1000
gr17	2085	2181	1.046043	510	1700
gr21	2707	3146	1.162172	630	2100
gr24	1272	1338	1.051887	720	2400
fri26	937	955	1.01921	780	2600
swiss42	1273	1589	1.248233	1260	4200
Att48	10628	12,915	1.215186	1440	4800
gr48	5056	5710	1.129351	1440	4800
eil51	426	546	1.28169	1530	5100
Berlin52	7542	9,024	1.1965	1560	5200
st70	675	910	1.348148	2100	7000
eil76	538	712	1.32342	2280	7600
br17	39	39	1	510	1700
Ftv35	1473	1668	1.132383	1080	3600
p43	5620	5657	1.006584	1290	4300
		Averages	1.144054	1162	3873

Table 6. GTTS Results with Intensification and Diversification

Problem	$T(p_{Optimal})$	$T(p_{Best})$	r	General Iterations	Intensify Iterations
Ohio10	678	678	1	300	1000
gr17	2085	2181	1.046043	510	1700
gr21	2707	2858	1.055781	630	2100
gr24	1272	1349	1.060535	720	2400
fri26	937	941	1.004269	780	2600
swiss42	1273	1427	1.120974	1260	4200
Att48	10628	11,239	1.05749	1440	4800
gr48	5056	5756	1.138449	1440	4800
eil51	426	522	1.225352	1530	5100
Berlin52	7542	8,649	1.146778	1560	5200
st70	675	895	1.325926	2100	7000
eil76	538	675	1.254647	2280	7600
br17	39	39	1	510	1700
Ftv35	1473	1620	1.099796	1080	3600
p43	5620	5652	1.005694	1290	4300
		Average	1.102782	1162	3873

In summary, the tables above present the Java GTTS algorithm results with a specific focus on the solution quality r and execution time. The chapter concludes with some analytical conclusions.

4.3 Analytical Conclusions

This section presents conclusions drawn from the GTTS results published in the tables above. First, a discussion on the effects of intensification and diversification on the solution quality r follows. Then, a discussion about the difference between intensification neighborhoods $N_1(p)$ and $N_2(p)$ and tabu tenures $t_1 = 0.10n$ and $t_2 = 0.20n$ is presented.

It appears from the tables in Section 4.2 that the GTTS intensification and diversification strategies improve the solution quality r. However, a 95 percent

confidence interval is formed to ensure our intuition. Table 7 below shows the average improvement in r when diversification, intensification, and intensification and diversification are employed. The normality assumption holds for r in all cases.

Table 7. Average Improvement in r with Diversification and Intensification

	Average Improvement	**95% Confidence Interval**
Average improvement in r from Diversification	-0.0865	(-0.154, -0.019)
Average improvement in r from Intensification	-0.109	(-0.173, -0.045)
Average improvement in r from Intensification and Diversification together	-0.150	(-0.237, -0.063)

As an illustration, suppose $r = 1.285$ for some general tabu search solution. Then we are 95 percent confident that r will improve (decrease) by some value within the interval (-0.063, -0.237) once intensification and diversification is employed, that is, $r \in (1.048, 1.22)$. Clearly, the GTTS algorithm's diversification and intensification strategy significantly improves the solution measure r.

The search neighborhood and tabu tenure effects the solution quality. Table 8 below presents some results to contrast the two search neighborhoods and Table 9 contrasts tabu tenures t_1 and t_2 where $t_1 = 0.10n$ and $t_2 = 0.20n$.

Table 8. Comparison of Intensification Neighborhoods N_1(p) and N_2(p)

Problem	Execution Time (sec.) (N_1(p))	r for N_1(p) and t_1	Execution Time (sec.) (N_2(p))	r for N_2(p) and t_1
Ohio10	3.13	1	2.61	1
gr17	7.07	1.04604	7.51	1.09496
gr21	42.64	1.19061	14.90	1.16217
gr24	76.61	1.05189	25.28	1.08726
fri26	110.67	1.02134	30.39	1.01921
Swiss42	816.52	1.41477	201.15	1.24823
Att48	1543.68	1.23513	364.75	1.24718
gr48	1549.49	1.17840	372.32	1.18968
eil51	2340.27	1.61972	433.75	1.36854
Berlin52	2768.87	1.30907	799.25	1.22713
st70	11,902.25	1.59259	1880.50	1.34815
eil76	17,856.00	1.45167	2378.08	1.32900
br17	18.80	1	7.22	1
Ftv35	388.81	1.23082	110.73	1.21860
p43	1005.65	1.01103	225.43	1.00658
Averages	2695.36	1.22354	456.92	1.16978

Table 9. Comparison of Tabu Tenure t_1 and t_2

Problem	r for t_1 and N_2(p)	r for t_2 and N_2(p)
Ohio10	1	1
gr17	1.08825	1.04604
gr21	1.05578	1.27078
gr24	1.09984	1.09984
fri26	1.01920	1.00427
Swiss42	1.37471	1.12097
Att48	1.05749	1.17110
gr48	1.19739	1.13845
eil51	1.22770	1.22535
Berlin52	1.14678	1.18404
st70	1.34667	1.32592
eil76	1.25465	1.32340
br17	1	1
Ftv35	1.22675	1.09980
p43	1.00658	1.00569
Averages	1.14011	1.13438

Forming a 95 percent confidence interval about the average improvement gives:

Table 10. Average Improvement in r when using N_2(p) or t_2.

	Average Improvement	95% Confidence Interval
Average improvement in r for using N_2(p)	-0.05376	(-0.108, 0)
Average improvement in r for using tenure t_2	-0.00574	(-0.06329, 0.05180)

Tables 8 and 10 support the hypothesis that N_2(p) is a better intensification neighborhood. The execution time of N_2(p) compared with N_1(p) demonstrates the impact of neighborhoods on the efficiency of the algorithm. The execution time for N_1(p) is nearly five times that of N_2(p). Statistically, N_2(p) also produces better solutions. Tables 9 and 10 show that there is no statistical significance in choosing tenure t_1 or t_2, at least when N_2(p) is used.

In summary, the GTTS intensification and diversification strategies show significant improvement in the general solution quality r, and N_2(p) is a much more efficient intensification neighborhood.

Chapter 5. Conclusions and Future Research

5.1 Introduction

This chapter concludes this GTTS research. The research conclusions are presented along with its contributions. After presenting the research conclusions and contributions, future research topics in GTTS are provided.

5.2 Research Conclusions and Contributions

This research demonstrates the first GTTS Java algorithm. Colletti (1999) presents the theory required for GTTS implementation and provides some pseudo code for a GTTS algorithm. However, this research demonstrates the first coded algorithm with empirical results.

The GTTS algorithm offers insight into the benefit of using group theory to implement a tabu search methodology. An important goal and claim of many heuristic search methods, in addition to being timely, is that they are simple to implement and understand. However, several tabu search TSP implementations within the literature, although they produce excellent results, are far from simple. "How" and "why" authors define their moves, intensification, and diversification strategies are cumbersome and lengthy, often too lengthy to present with the required detail. The GTTS algorithm demonstrates a simple, yet powerful, way to define moves as well as the intensification and diversification strategies. This is done using a simple algebraic equation, which is possible because the solutions for the TSP are elements of the algebraic group S_n.

The baseline of GTTS is presented in Figure 7 below. This baseline is used to

1. Given incumbent solution p,
2. select new solution $q = p^m$ and then
3. set p = q and repeat.

Figure 7. GTTS Algorithm Baseline

define GTTS concepts. Consider the general GTTS concepts of move and neighborhood definition, intensification, and diversification. In step 1, GTTS begins the search with p = (0,1,2, . . .,*n*-1), $p = p_j$ if intensifying, where p_j is the j^{th} best solution found in the initial search, p = the $(0.005)\mathbf{I}^{th}$ consecutive non-improving move, if diversifying. In step 2, we select q for m = (*a*,*b*), or m = (*a*,*b*,*c*) if intensifying, such that *T*(q) is minimized for all q $\in$ *N*(p) subject to the tabu and aspiration criteria. Here, $a, b, c \in A = \{1,2,\ldots,n-1\}$. For diversification, $m = (a_1, a_2, \ldots, a_m)$, where $m = (0.15)n$ and a_k is the k^{th} least moved node. The GTTS algorithm essentially repeats the baseline procedure for p, q, and m as described above. Simply stated, the entire GTTS methodology described in this research is explained in one paragraph. Clearly, the key to this simplicity is the simple equation $q = p^m = m^{-1}pm$, i.e., the conjugate of p by m, which is a significant contribution of group theory in describing tabu search.

Notably, the GTTS implementation presented in this thesis is very basic. In fact, the two-letter rearrangement, or swap, is a very traditional move for TSP tabu search implementations within the literature. However, conjugation is not limited to two, three, or even four letter rearrangements, but is only limited by the problem size *n* since $m \in S_n$. Thus, any tabu search implementation whose solutions are found by rearranging its elements, or letters, may be defined in a simple and brief manner using conjugation. Simple implementations usually lead to a better understanding of the overall metaheuristic's behavior.

The application of GTTS is still in its infancy and requires more testing and research. Like all new concepts, GTTS must overcome its own hurdles before it will be accepted and implemented. Two immediate hurdles for GTTS are first, the general unfamiliarity with group theoretic concepts, and second, the lack of computational group theory codes. In order to use, understand, and implement GTTS, the researcher must become familiar with the general concepts of group theory. Although the concepts of group theory are very simple, when taken as a whole they become very overwhelming and abstract. However, the full power of GTTS relies on understanding the basic properties of S_n and then applying these properties to develop better tabu search moves and neighborhoods. In order to apply the properties of S_n, computer codes need refined and developed for computational group theory. At this point, the GAP software referenced in Chapter 3 is essentially the only existing comprehensive code for computational group theory, but a more expansive library is needed, particularly for the group S_n. Once computer code is readily available and group theory concepts are more familiar, GTTS will be implemented more readily and become a robust and powerful tool for finding quality and timely solutions to combinatorial optimization problems such as the TSP class of problems. Thus, GTTS requires a vast amount on continuing research. The following section discusses such research.

5.3 Future Research

The GTTS algorithm presented in this thesis uses conjugation, which is one simple group theoretic concept, but there are several other group theoretic concepts worth

exploring to further GTTS research. Colletti (1999) suggests several group theoretic concepts worth implementing in the GTTS algorithm.

The success of GTTS relies heavily on the search neighborhood structure. Clearly, the neighborhood structure depends on the move definition, but the move alone does not eliminate the task of finding the ideal search neighborhood. For example, in Chapter 3, the neighborhood for the two-letter rearrangement move considered all possible transpositions m, but for the three-letter rearrangement move, it became impractical to consider all possible 3-cycle moves. Therefore, what is the best neighborhood structure? Ideally, we seek quality neighborhoods that are relatively small and easy to evaluate. Colletti (1999) offers the use of subgroups and cosets (see Section 2.4) to generate neighborhoods for an incumbent solution p. Generating subgroups and cosets require sophisticated computational group theoretic code, but they provide a strong theoretical basis for neighborhood building.

Statistical analysis and group theory may be combined to determine the parameter settings, such as the tabu tenure, number of search iterations, and number of allowed non-improving moves, that will produce the best solutions. Chapter 4 presents some differences in solution quality as the tabu tenure and neighborhood structure change, but a more formal statistical and group theoretical look may provide insights that are more useful.

The GTTS algorithm presented in this research may be improved and extended in several ways. First, in order to solve real world problems like those faced by AMC; the code must be extended from the TSP to solve the other TSP class of problems, such as

the mTSP, mTSPTW, and VRP. Second, the overall efficiency and sophistication of the code may be improved.

As the TSP grows in complexity to include the mTSP, mTSPTW, and VRP, the feasible tour structure moves from an *n*-cycle to a product of *m*-cycles. Thus, problems like AMC's airlift routing problem require not only a rearrangement of nodes, but a partitioning of nodes too. Permutation multiplication, by the use of templates, provides this partitioning as simply as conjugation provides a rearrangement of nodes. Recall from Section 2.4.6 that templates are permutations that partition another permutation into either more disjoint cycles (a splitting template) or fewer disjoint cycles (a welding template). Thus, the GTTS algorithm move may be extended to include moves m, where pm = q, that is, a template m partitions p to yield q. Just as conjugation is the foundation for defining node rearrangement moves, templates are the foundation for defining partitioning moves. By the use of templates, the GTTS algorithm currently suited for the basic TSP naturally evolves to handling problems that are more complex.

The overall code efficiency and sophistication may be improved in several ways. One way is to implement a reactive tabu tenure that allows the tenure to adjust to the quality of search. Using hash tables to track solution collisions and tabu moves should increase the solution quality and algorithm efficiency. Adding time window constraints is another improvement required to solve AMC's airlift routing problem. Whether group theory provides added efficiency is worthy of extensive research from a computer science perspective.

GTTS is exciting and promising research. The need to solve large practical problems in a timely manner warrants continued research, both theoretically and computationally, into GTTS.

Appendix A. Java Documentation

A.1 Code Description

The GTTS algorithm is implemented in the Java programming language. Java is a high-level object-oriented language. Thus, a program is made up of several classes, which when instantiated, become objects. The GTTS algorithm employs several objects. A major advantage of object-oriented coding is code sharing and reuse. The GTTS algorithm takes advantage of the tabu search framework developed by Harder (2000). Harder's framework provides an engine to run any general tabu search implementation. See Harder (2000) for details and the source code for the general tabu search objects. This appendix contains the source code specific for this research, i.e., for the specific GTTS implementation.

A.2 Source Code

The section contains the GTTS Java source code for fourteen Java classes. Each class is coded with in-line comments describing its purpose and function. In Java, a statement preceded by // or /* indicates a comment. The first ten classes below combine together to define the move, tabu and frequency list, intensification strategy, and diversification strategy. The next three classes are for operating on permutations, i.e., these classes multiply, invert, and conjugate permutations. The final class is a data object used for importing the cost matrix for each problem instance into Java from an *Access* database. The source code follows with each class being preceded by an underlined title.

Execution Class

```
/*****************************************************************************
* This class executes the group theoretic tabu search (GTTS) algorithm.  The initial
* inputs are defined within this class.
*****************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPExecute implements Runnable,
                                   NewCurrentSolutionListener,
                                   NewBestSolutionListener,
                                   UnimprovingMoveListener
{
   // The variables below are used for diversification purposes.
   int numberOfUnimprovingMoves = 0;
   int IterationCounter = 0;
   int maxUnimprovingMoves;
   int diversificationMoveSize;
   static int diversificationCount = 0;

   // The Distance Matrix determines the tourlength T(p).
   static int [][] DistanceMatrix;

   // The following array keeps a list of the elite solutions.  The array is initialized
   // to save to top 5 solutions, but may be initialized to any desired length.
   static TSPSolution[] eliteSolutions = new TSPSolution[5];

   // The objectiveFunction variable is made static so that it may be used globally.
   // Specifically, it is used for diversification.
   public static TSPObjectiveFunction objectiveFunction;

   // The variable "numberNodes" determines the size of the problem.  The solution space
   // is the Symmetric Group on n letters (S sub n) where n = numberNodes.  The variable
   // is static so that it may be referenced in other classes.  For example, the number
   // of nodes is used to define the neighborhood size in the TSPMoveManager Class.
   static int numberNodes;

   // Variables for the number of iterations to perform
   int GeneralTSIterations;
   int IntensificationIterations;

   // Variables for the tabu tenure
   int GeneralTSTenure;
   int IntensificationTenure;

   // The variable "thread" specifies whether one or two threads will be used during
   // the search.  Thus, for machines with multiprocessors, the user may specify
   // "threads = 2".
   int threads = 1;

   // Constructor Method
   public TSPExecute( String[] args )
   {
```

```
    if( args.length == 2 )
    {
      GeneralTSIterations = new Integer( args[0] ).intValue();
      threads = new Integer( args[1] ).intValue();
    }
}

public static void main( String[] args )
{
   Thread t = new Thread ( new TSPExecute(args) );
   t.start();
}

public void run()
{
   DistanceMatrix = DataSets.cost;

   // The array below is the initial starting solution for the search.  Since
   // this algorithm is a demonstration of the group theoretic metaheuristic theory
   // developed by Colletti (1999), the initial starting solution is not important.
   int[] x = new int[48];
   for( int k = 0; k < x.length; k++)
      x[k] = k;

   numberNodes = x.length;

   // Input for the number of iterations to perform , the tabu tenure, and the maximum number of
   // consecutive unimproving moves.
   GeneralTSIterations = 30*numberNodes;
   IntensificationIterations = 20*numberNodes;
   GeneralTSTenure = Math.max(2,(int)(0.10*numberNodes);
   IntensificationTenure = GeneralTSTenure;
   maxUnimprovingMoves = Math.max(5,(int)(.005*GeneralTSIterations));
   diversificationMoveSize = Math.max(5, (int)(0.15*numberNodes));

   TSPSolution initialSolution = new TSPSolution( x );
   objectiveFunction = new TSPObjectiveFunction();
   TSPConstraintPenalties constraintPenalties = new TSPConstraintPenalties();
   TSPTabuList tabuList = new TSPTabuList( GeneralTSTenure );
   TSPMoveManager moveManager = new TSPMoveManager();

   // The Engine class is the workhorse of the tabu search algorithm.  Harder (2000)
   // developed this code for a general tabu search algorithm. The Boolean parameter
   // "false" indicates a minimization problem.
   Engine engine = new Engine(initialSolution, tabuList,
                     objectiveFunction, constraintPenalties,
                     moveManager, false, threads );

   // This code implements listeners for diversification and intensfication purposes.
   engine.addNewCurrentSolutionListener( this );
   engine.addNewBestSolutionListener( this );
   engine.addUnimprovingMoveListener( this );

   engine.startSolving( GeneralTSIterations );
```

```
    engine.waitToFinish();
    System.out.println( "Best General Tabu Search Solution : " + engine.getBestSolution() );

    // The following code is the intensification code.  It performs a new search starting
    // at the elite solutions found by "engine" above.
    TSPSolution intensify1 = eliteSolutions[0];
    TSPIntensificationTabuList intensify1tabuList = new TSPIntensificationTabuList(
IntensificationTenure );
    TSPIntensificationMoveManager intensify1moveManager = new TSPIntensificationMoveManager();
    Engine intensificationEngine1 = new Engine(intensify1, intensify1tabuList,
                                objectiveFunction, constraintPenalties,
                                intensify1moveManager, false, threads );
    intensificationEngine1.startSolving( IntensificationIterations );
    intensificationEngine1.waitToFinish();
    System.out.println( "Intensification 1 Best Solution : " +
                intensificationEngine1.getBestSolution() );

    TSPSolution intensify2 = eliteSolutions[1];
    TSPIntensificationTabuList intensify2tabuList = new TSPIntensificationTabuList(
IntensificationTenure );
    TSPIntensificationMoveManager intensify2moveManager = new TSPIntensificationMoveManager();
    Engine intensificationEngine2 = new Engine(intensify2, intensify2tabuList,
                                objectiveFunction, constraintPenalties,
                                intensify2moveManager, false, threads );
    intensificationEngine2.startSolving( IntensificationIterations );
    intensificationEngine2.waitToFinish();
    System.out.println( "Intensification 2 Best Solution : " +
                intensificationEngine2.getBestSolution() );

    TSPSolution intensify3 = eliteSolutions[2];
    TSPIntensificationTabuList intensify3tabuList = new TSPIntensificationTabuList(
IntensificationTenure );
    TSPIntensificationMoveManager intensify3moveManager = new TSPIntensificationMoveManager();
    Engine intensificationEngine3 = new Engine(intensify3, intensify3tabuList,
                                objectiveFunction, constraintPenalties,
                                intensify3moveManager, false, threads );
    intensificationEngine3.startSolving( IntensificationIterations );
    intensificationEngine3.waitToFinish();
    System.out.println( "Intensification 3 Best Solution : " +
                intensificationEngine3.getBestSolution() );

    TSPSolution intensify4 = eliteSolutions[3];
    TSPIntensificationTabuList intensify4tabuList = new TSPIntensificationTabuList(
IntensificationTenure );
    TSPIntensificationMoveManager intensify4moveManager = new TSPIntensificationMoveManager();
    Engine intensificationEngine4 = new Engine(intensify4, intensify4tabuList,
                                objectiveFunction, constraintPenalties,
                                intensify4moveManager, false, threads );
    intensificationEngine4.startSolving( IntensificationIterations );
    intensificationEngine4.waitToFinish();
    System.out.println( "Intensification 4 Best Solution : " +
                intensificationEngine4.getBestSolution() );

    TSPSolution intensify5 = eliteSolutions[4];
```

```
    TSPIntensificationTabuList intensify5tabuList = new TSPIntensificationTabuList(
IntensificationTenure );
    TSPIntensificationMoveManager intensify5moveManager = new TSPIntensificationMoveManager();
    Engine intensificationEngine5 = new Engine(intensify5, intensify5tabuList,
                                objectiveFunction, constraintPenalties,
                                intensify5moveManager, false, threads );
    intensificationEngine5.startSolving( IntensificationIterations );
    intensificationEngine5.waitToFinish();
    System.out.println( "Intensification 5 Best Solution : " +
                intensificationEngine5.getBestSolution() );

    // Outputs for the execution time
    System.out.println( "General Solution time (sec): " + ( engine.getLastSolveMillis() / 1000. ) );
    System.out.println( "Intensification 1 Solution time (sec): " +
                ( intensificationEngine1.getLastSolveMillis() / 1000. ) );
    System.out.println( "Intensification 2 Solution time (sec): " +
                ( intensificationEngine2.getLastSolveMillis() / 1000. ) );
    System.out.println( "Intensification 3 Solution time (sec): " +
                ( intensificationEngine3.getLastSolveMillis() / 1000. ) );
    System.out.println( "Intensification 4 Solution time (sec): " +
                ( intensificationEngine4.getLastSolveMillis() / 1000. ) );
    System.out.println( "Intensification 5 Solution time (sec): " +
                ( intensificationEngine5.getLastSolveMillis() / 1000. ) );

    // Outputs number of iterations and tabu tenure.
    System.out.println( "Number of General Iterations: " + GeneralTSIterations );
    System.out.println( "Number of Intensification Iterations: " + IntensificationIterations );
    System.out.println( "Tabu Tenure is: "+GeneralTSTenure);
    System.out.println( "Number of Diversification moves performed: " + diversificationCount );
  } // end run method

  public void newCurrentSolution( TabuEvent e )
  {
    IterationCounter++;
  }

  // This method is fired whenever a new best solution is found. This code update the elite
  // solutions list.
  public void newBestSolution( TabuEvent e )
  {
    TSPSolution bestSolution = (TSPSolution) ((Engine)e.getSource()).getBestSolution();
    for(int n = 4; n > 0; n--)
    {
      eliteSolutions[n] = eliteSolutions[n-1];
    }
    eliteSolutions[0] = bestSolution;

  }

  // This method is fired whenever a non-improving move is made.  The code in this method
  // implements the diversification strategy.
  public void unimprovingMoveMade( TabuEvent e )
  {
    numberOfUnimprovingMoves++;
```

```
if(numberOfUnimprovingMoves != IterationCounter)
{
   numberOfUnimprovingMoves = 0;
   IterationCounter = 0;
}

else if(numberOfUnimprovingMoves == maxUnimprovingMoves)
{
   numberOfUnimprovingMoves = 0;
   diversificationCount++;
   // m is the diversification move.
   int[] m = new int[diversificationMoveSize];

   // The following code searches the frequency array to find the least
   // swapped nodes and then populates m.
   int min = TSPTabuList.frequency[0];
   for(int i = 1; i < TSPTabuList.frequency.length; i++)
   {
      if(TSPTabuList.frequency[i]< min)
         min = TSPTabuList.frequency[i];
   }
   int k = 0;
   while(m[m.length-1] == 0)
   {
      for(int t = 0; t < TSPTabuList.frequency.length; t++)
      {
         if(TSPTabuList.frequency[t] == min)
         {
            m[k] = t + 1;
            if(k == m.length-1)
               break;
            k++;
         }
      }
      min += 1;
   }
   //for(int a = 0; a < m.length; a++)
   //{
    //  System.out.print(m[a]+" ");
   //}
   //System.out.println("");

   // The code below gets the current incumbent solution p and conjugates it with
   // m to yield q, the new incumbent solution.
   TSPSolution currentSolution = (TSPSolution) ((Engine)e.getSource()).getCurrentSolution();
   int[] p = new int[numberNodes];
   for(int v = 0; v < p.length; v++)
   {
      p[v] = currentSolution.x[v];
   }
   int[] q;
   q = PermutationConjugation.conjugate2(p,m);
   for(int u = 0; u < q.length; u++)
```

```
        {
          currentSolution.x[u] = q[u];
        }
        double val = objectiveFunction.evaluate(currentSolution);
        currentSolution.setObjectiveValue( val );
      }
    } // ends unimprovingMoveMade method
}  // end class TSPExecute
```

Solution Class

```
/**********************************************************************
* This class defines the solution form for the tabu search framework.
* The solution for the GTTS algorithm is an array of length n that
* represents a n-cycle permutation.
**********************************************************************/
import net.usa.rharder.tabusearch.*;

public class TSPSolution extends Solution
{
   int[] x;

   public TSPSolution( int[] passedArray )
   {
      x = new int[ passedArray.length ];
      for( int i = 0; i < x.length; i++ )
         x[i] = passedArray[i];
   }

   public int size()
   {
      return x.length;
   }

   public Object clone()
   {
      return new TSPSolution( x );
   }

   public String toString()
   {
      // This method outputs the objective and tour for the best
      // solution found.
      String self = "Distance = " + (this.getValue()) + "\n";
      //for( int i = 0; i < x.length; i++ )
         //self += "Go to node " + (x[i]) + " and then" + "\n";
      return self;
   }
}  // end class TSPSolution
```

Objective Function Class

```
/*****************************************************************************
* This class evaluates the objective function (tour length) for all moves.
* The evaluate method uses the tourlength equations given by Colletti (1999).
* (not yet programmed).
*****************************************************************************/
import net.usa.rharder.tabusearch.*;

public class TSPObjectiveFunction extends TabuFunction
{
   public double evaluate( Solution soln )
   {
      TSPSolution solution = (TSPSolution) soln;
      // Evaluate the solution
      double value = 0;
      for( int i = 0; i < TSPExecute.numberNodes - 1; i++ )
         value += TSPExecute.DistanceMatrix[solution.x[i]][solution.x[i+1]];
      value += TSPExecute.DistanceMatrix[solution.x[TSPExecute.numberNodes-1]][solution.x[0]];
      return value;
   }

}  // end class TSPObjectiveFunction
```

Move Manager Class

```
/*************************************************************************
* This class manages the search neighborhood at each iteration. At each
* iteration, each move is stored in an array and then the array of moves
* is sent to the engine for evaluation.
*************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPMoveManager extends MoveManager
{
   public Move[] getAllMoves( Solution soln )
   {
      int counti, letter1, letter2;
      // In the conjugation p^m = q where m=(a,b), letter1 = a, and
      // letter2 = b.  The variable counti is the index reference for
      // the array of all moves, i.e, counti = 0 indexes the first move
      // for that iteration.  Since p is of size n and we hold the
      // first element fixed, there are [(n-1)(n-2)] neighbors at
      // each iteration.  However, since m = (a,b) = (b,a) = mINV, then
      // the neighborhood size reduces to [(n-1)(n-2)]/2. Hence, counti
      // ranges from zero to ([(n-1)(n-2)]/2) - 1.
      counti = 0;
      letter1 = 0;
      letter2 = 0;
      TSPSolution solution = (TSPSolution) soln;
      TSPMove[] allMoves = new TSPMove[((TSPExecute.numberNodes-2)*
                           (TSPExecute.numberNodes-1))/2];

      // The variable j below represents a position in permutation p, or index
      // of the solution array.  The letter in position j is letter1 above.
      // letter2 is in position j+i+1.  Thus, the variable i aides
      // in identifying the second position and counts the number of times
      // letter1 = a.  For example, if the current solution is the array (cycle)
      // p = (0,1,2,3,4,5) and j = 1, then m = (1,i+2) where i ranges from 0 to
      // n - (j+2) = 6 - 3 = 3. So the first four moves are m = (1,2), m = (1,3),
      // m = (1,4), and m = (1,5).
      for( int j = 1; j < TSPExecute.numberNodes-1; j++)
      {
         letter1 = solution.x[j];
         for( int i = 0; i < (TSPExecute.numberNodes-1) - j; i++)
         {
            letter2 = solution.x[j+i+1];
            allMoves[counti] = new TSPMove( letter1, letter2, j, i);
            counti++;
         }
      }
      return allMoves;
   }
} // end class TSPMoveManager
```

Intensification Move Manager Class

```
/*************************************************************************
* This class manages the intensification search neighborhood at each
* intensification iteration. At each iteration, each move is stored in an
* array and then the array of moves is sent to the engine for evaluation.
*************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPIntensificationMoveManager extends MoveManager
{
  //Intensification Method One
  /*
  public Move[] getAllMoves( Solution soln )
  {
    int counti, letter1, letter2, letter3;
    // In the conjugation p^m = q where m=(a,b,c), letter1 = a,
    // letter2 = b, and letter3 = c.  The variable counti is the index
    // reference for the array of all moves, i.e, counti = 0 indexes the
    // first move for that iteration.  Since p is of size n and we hold the
    // first element fixed, the neighborhood for p is very large as n
    // gets large.  Each iteration only considers all 3-cycles beginning
    // at the letter in the second, third, fourth, and fifth positions of
    // the current tour. Thus, there are [(n-3)(n-3)+(n-5)(n-5)] neighbors at
    // each iteration. Hence, counti ranges from zero to
    // ([(n-3)(n-3)+(n-5)(n-5)] - 1).
    counti = 0;
    letter1 = 0;
    letter2 = 0;
    letter3 = 0;
    TSPSolution solution = (TSPSolution) soln;
    TSPIntensificationMove[] allMoves = new TSPIntensificationMove
                          [((TSPExecute.numberNodes-3)*
                          (TSPExecute.numberNodes-3))+
                          (TSPExecute.numberNodes-5)*
                          (TSPExecute.numberNodes-5)];

    // The variable j below represents a position in permutation p, or index
    // of the solution array.  The letter in position j is letter1 above.
    // letter2 is in position j+i+1.  letter3 is in position j+i+k+2.

    for( int j = 1; j < 5; j++)
    {
      letter1 = solution.x[j];
      for( int i = 0; i < (TSPExecute.numberNodes-2) - j; i++)
      {
        letter2 = solution.x[j+i+1];
        for( int k = 0; k < (TSPExecute.numberNodes-2) - (j+i); k++)
        {
          letter3 = solution.x[j+i+k+2];
          allMoves[counti] = new TSPIntensificationMove( letter1, letter2, letter3, j, i, k);
          counti++;
        }
```

```
        }
      }
      return allMoves;
    } */

    // Intensification Method Two
    public Move[] getAllMoves( Solution soln )
    {
      int count, letter1, letter2, letter3;
      // In the conjugation p^m = q where m=(a,b,c), letter1 = a,
      // letter2 = b, and letter3 = c. The variable counti is the index
      // reference for the array of all moves, i.e, counti = 0 indexes the
      // first move for that iteration. Since p is of size n and we hold the
      // first element fixed, the neighborhood for p is very large as n
      // gets large. There are [(n-3)(n-2)]/2 neighbors at each iteration.
      count = 0;
      letter1 = 0;
      letter2 = 0;
      letter3 = 0;
      TSPSolution solution = (TSPSolution) soln;
      TSPIntensificationMove[] allMoves = new TSPIntensificationMove
                                [((TSPExecute.numberNodes-3)*
                                 (TSPExecute.numberNodes-2))/2];

      // The variable j below represents a position in permutation p, or index
      // of the solution array. The letter in position j is letter1 above.
      // letter2 is in position j+i+1. letter3 is in position j+i+2.

      for( int j = 1; j < TSPExecute.numberNodes-2; j++)
      {
        letter1 = solution.x[j];
        for( int i = 0; i < (TSPExecute.numberNodes-2) - j; i++)
        {
          letter2 = solution.x[j+i+1];
          letter3 = solution.x[j+i+2];
          allMoves[count] = new TSPIntensificationMove( letter1, letter2, letter3, j, i);
          count++;
        }
      }
      return allMoves;
    }
  } // end class TSPIntensificationMoveManager
```

Move Class

```
/*****************************************************************************
* This class defines the tabusearch move.  The move for this algorithm is the two-letter
* rearrangement (a swap).  That is, p^m = mINV*p*m = q where p is the current solution,
* m is a transposition, and q is the resulting tour. If m = (a,b), then m yields a swap move
* which is a 4-opt move if a and b and non-adjacent letters in p and a 3-opt move if a and
* b are adjacent in p.
*****************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPMove extends Move
{
   int i, j, letter1, letter2;

   public TSPMove( int movenumber1, int movenumber2, int index, int positions)
   {
      // These inputs are passed from the TSPMoveManager to here to
      // perform the individual move.  See the TSPMoveManager class for a
      // description of the variables below.
      j = index;
      i = positions;
      letter1 = movenumber1;
      letter2 = movenumber2;
   }

   public void operateOn( Solution soln )
   {
      // This method performs the move on the incumbent solution "soln"
      // using the PermutationConjugation class.
      TSPSolution solution = (TSPSolution) soln;

      int m[] = {solution.x[j], solution.x[j+i+1]};
      int p[] = new int[TSPExecute.numberNodes];
      for( int k = 0; k < solution.x.length; k++ )
         p[k] = solution.x[k];
      int q[];
      q = PermutationConjugation.conjugate2(p,m);
      solution.x[j] = q[j];
      solution.x[j+i+1] = q[j+i+1];
   }

   public void undoOperation( Solution soln )
   {
      // This method reverses the conjugation to preserve the incumbent
      // solution for the next move operation.
      operateOn( soln );
   }

   // The following methods supply letter1 and letter3 to the tabu
   // list. See the TSPTabuList class.
   public int getID1()
   {
```

```
    return letter1;
  }

  public int getID2()
  {
    return letter2;
  }

  public boolean conflictsWith( Move move )
  {
    boolean conflicts = true;
    return conflicts;
  }
} // end class TSPMove
```

Intensification Move Class

```
/*****************************************************************************
* This class defines the intensification move.  The intensification move for this
* algorithm is different from the general tabu search move.  The move for intensification
* is the three-letter rearrangement move.  That is, p^m = mINV*p*m = q where p is the
* current solution, m is a 3-cycle, and q is the resulting tour. Thus, m = (a,b,c).
*****************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPIntensificationMove extends Move
{
  // Intensification Method One
  /*
  int i, j, k, letter1, letter2, letter3;

  public TSPIntensificationMove( int movenumber1, int movenumber2, int movenumber3, int index, int
positions, int positions2)
  {
    // These inputs are passed from the TSPIntensificationMoveManager for this class
    // to perform the individual move.  See the TSPIntensificationMoveManager class for a
    // description of the variables below.
    j = index;
    i = positions;
    k = positions2;
    letter1 = movenumber1;
    letter2 = movenumber2;
    letter3 = movenumber3;
  }

  public void operateOn( Solution soln )
  {
    // This method performs the move on the incumbent solution "soln"
    // using the PermutationConjugation class.
    TSPSolution solution = (TSPSolution) soln;

    int m[] = {letter1, letter2, letter3};
    int p[] = new int[TSPExecute.numberNodes];
    for( int s = 0; s < solution.x.length; s++ )
      p[s] = solution.x[s];
   int q[];
    q = PermutationConjugation.conjugate2(p,m);
    for(int v = 0; v < solution.x.length; v++)
    {
      solution.x[v] = q[v];
    }
  }

  public void undoOperation( Solution soln )
  {
    // This method reverses the conjugation to preserve the incumbent
    // solution for the next move operation.
    TSPSolution solution = (TSPSolution) soln;
```

```
    int [] mINV = {letter3, letter2, letter1};
    int q[] = new int[TSPExecute.numberNodes];
    for( int s = 0; s < solution.x.length; s++ )
      q[s] = solution.x[s];
    int p[];
    p = PermutationConjugation.conjugate2(q,mINV);
    for(int w = 0; w < solution.x.length; w++)
    {
      solution.x[w] = p[w];
    }
  } */

  // Intensification Method Two
  int i, j, letter1, letter2, letter3;

  public TSPIntensificationMove( int movenumber1, int movenumber2, int movenumber3, int index, int
positions)
  {
    // These inputs are passed from the TSPIntensificationMoveManager for this class
    // to perform the individual move.  See the TSPIntensificationMoveManager class for a
    // description of the variables below.
    j = index;
    i = positions;
    letter1 = movenumber1;
    letter2 = movenumber2;
    letter3 = movenumber3;
  }

  public void operateOn( Solution soln )
  {
    // This method performs the move on the incumbent solution "soln"
    // using the PermutationConjugation class.
    TSPSolution solution = (TSPSolution) soln;

    int m[] = {letter1, letter2, letter3};
    int p[] = new int[TSPExecute.numberNodes];
    for( int s = 0; s < solution.x.length; s++ )
      p[s] = solution.x[s];
    int q[];
    q = PermutationConjugation.conjugate2(p,m);
    for(int v = 0; v < solution.x.length; v++)
    {
      solution.x[v] = q[v];
    }
  }

  public void undoOperation( Solution soln )
  {
    // This method reverses the conjugation to preserve the incumbent
    // solution for the next move operation.
    TSPSolution solution = (TSPSolution) soln;

    int [] mINV = {letter3, letter2, letter1};
```

```
      int q[] = new int[TSPExecute.numberNodes];
      for( int s = 0; s < solution.x.length; s++ )
         q[s] = solution.x[s];
      int p[];
      p = PermutationConjugation.conjugate2(q,mINV);
      for(int w = 0; w < solution.x.length; w++)
      {
         solution.x[w] = p[w];
      }
   }

   // The following methods supply letter1, letter2, and letter3 to the tabu
   // list. See the TSPIntensificationTabuList class.
   public int getID1()
   {
      return letter1;
   }

   public int getID2()
   {
      return letter2;
   }

   public int getID3()
   {
      return letter3;
   }

   public boolean conflictsWith( Move move )
   {
      boolean conflicts = true;
      return conflicts;
   }
} // end class TSPIntensificationMove
```

Tabu List Class

```
/*****************************************************************************
* This class stores and updates the recency and frequency information for the search.
* Recency is tracked by the tabu list and frequency is tracked by the frequency list.
* The tabu list records the most recent nodes that have been swapped, while the frequency
* list tracks the number of times each node is part of a swap.
*****************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPTabuList extends TabuList
{
   int tenure, swappednode1, swappednode2;
   int[] tabulist;
   static int[] frequency;

   public TSPTabuList( int pTenure )
   {
      tenure = pTenure;
      tabulist = new int[ 2*tenure ];
      frequency = new int[TSPExecute.numberNodes - 1];
   }

   public boolean allowMove( Move move, Solution solution )
   {
      boolean allow = true;
      int i = 0;
      while( (allow == true) && ( i < tabulist.length ) )
      {
         if( tabulist[i++] == ((TSPMove)move).getID1() )
            allow = false;
         if( tabulist[i++] == ((TSPMove)move).getID2() )
            allow = false;
      }
      return allow;
   }

   public void registerMoves( Move[] moves, Solution soln )
   {
      for( int i = 0; i < moves.length; i++ )
         registerMove( moves[i], soln );
   }


   public void registerMove( Move move, Solution solution )
   {
      // The following code adds the two swapped nodes to the tabu list
      for( int i = 0; i < tabulist.length-2; i++ )
         tabulist[i] = tabulist[i+2];
      tabulist[tabulist.length - 2] = ((TSPMove)move).getID1();
      tabulist[tabulist.length - 1] = ((TSPMove)move).getID2();

      // The following code updates the frequency list for the swapped nodes.
```

```
        swappednode1 = ((TSPMove)move).getID1();
        swappednode2 = ((TSPMove)move).getID2();
        frequency[swappednode1 - 1] = frequency[swappednode1 - 1] + 1;
        frequency[swappednode2 - 1] = frequency[swappednode2 - 1] + 1;
    }

    public String toString()
    {
        String self = "Tabu moves are ";
        for( int i = 0; i < tabulist.length; i++ )
            self += tabulist[i] + " ";
        return self;
    }  // end toString
}  // end TSPTabuList
```

Intensification Tabu List Class

```
/*****************************************************************************
* This class stores and updates the recency and frequency information for the intensification
* search. Recency is tracked by the tabu list and frequency is tracked by the frequency list.
* The tabu list records the most recent nodes that have been rearranged, while the frequency
* list tracks the number of times each node is part of a rearranement.
*****************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPIntensificationTabuList extends TabuList
{
   int tenure, swappednode1, swappednode2, swappednode3;
   int[] tabulist;
   int[] frequency;

   public TSPIntensificationTabuList( int pTenure )
   {
      tenure = pTenure;
      tabulist = new int[ 3*tenure ];
      frequency = new int[TSPExecute.numberNodes - 1];
   }

   public boolean allowMove( Move move, Solution solution )
   {
      boolean allow = true;
      int i = 0;
      while( (allow == true) && ( i < tabulist.length ) )
      {
         if( tabulist[i++] == ((TSPIntensificationMove)move).getID1() )
            allow = false;
         if( tabulist[i++] == ((TSPIntensificationMove)move).getID2() )
            allow = false;
      }
      return allow;
   }

   public void registerMoves( Move[] moves, Solution soln )
   {
      for( int i = 0; i < moves.length; i++ )
         registerMove( moves[i], soln );
   }

   public void registerMove( Move move, Solution solution )
   {
      // The following code adds the three swapped nodes to the tabu list
      for( int i = 0; i < tabulist.length-3; i++ )
         tabulist[i] = tabulist[i+3];
      tabulist[tabulist.length - 3] = ((TSPIntensificationMove)move).getID1();
      tabulist[tabulist.length - 2] = ((TSPIntensificationMove)move).getID2();
      tabulist[tabulist.length - 1] = ((TSPIntensificationMove)move).getID3();

      // The following code updates the frequency list for the swapped nodes.
```

```
        swappednode1 = ((TSPIntensificationMove)move).getID1();
        swappednode2 = ((TSPIntensificationMove)move).getID2();
        swappednode3 = ((TSPIntensificationMove)move).getID3();
        frequency[swappednode1 - 1] = frequency[swappednode1 - 1] + 1;
        frequency[swappednode2 - 1] = frequency[swappednode2 - 1] + 1;
        frequency[swappednode3 - 1] = frequency[swappednode3 - 1] + 1;

    }

    public String toString()
    {
      String self = "Tabu moves are ";
      for( int i = 0; i < tabulist.length; i++ )
        self += tabulist[i] + " ";
      return self;
    }  // end toString

}  // end TSPIntensificationTabuList
```

Constraint Penalty Class

```
/*****************************************************************************
* This class allows for constraints to be added to the GTTS algorithm,
* such as time windows, etc.  The engine requires a value to
* be returned from this class.  Thus, if no contstraints exist, then zero
* is returned.
*****************************************************************************/

import net.usa.rharder.tabusearch.*;

public class TSPConstraintPenalties extends TabuFunction
{
   public double evaluate( Solution soln )
   {
      TSPSolution solution = (TSPSolution) soln;
      double value = 0;
      return value;
   }
}  // end class TSPConstraintPenalties
```

Permutation Multiplication Class

```
/*************************************************************************
* This class takes two permutations represented as integer arrays and multiplies the two
* permutations (arrays). The multiply method returns a double array that
* represents the product as a standard matrix permutation.
*************************************************************************/

import java.lang.Math;

public class PermutationMultiplication
{
  public PermutationMultiplication()
  {
    // Constructor method
  }

  public static int [][] multiply(int x[], int y[])
  {
    /* Variables for determining the size of the permutation */
    int maxp, maxq, max;

    /* Variable for indexing loops */
    int i, j, k, l, m, n, o, s, t, u;

    /* These are the permutations to be multiplied */
    int p[] = x;
    int q[] = y;

    /* g and h are the above permutations in standard matrix format */
    int g[][];
    int h[][];

    /* r is the product of g*h = gh, which is not necessarily h*g = hg */
    int r[][];

    /* The following code determines the maximum integer in p or q which
      defines the size of the permutation. It does this by comparing each
      element in each array. */
    maxp = p[0];
    maxq = q[0];

    for( i = 1; i < p.length; i++ )
      if( p[i] > maxp )
      {
        maxp = p[i];
      }

    for( j = 1; j < q.length; j++ )
      if( q[j] > maxq )
      {
        maxq = q[j];
      }
```

```
max = Math.max(maxp, maxq);

/* Defines the permutations in standard matrix format */
g = new int[2][max];
h = new int[2][max];
r = new int[2][max];

/* The loops define the top row of each matrix permutation as
   1, 2, 3, . . ., "max" where "max" is the largest integer in p or q */
int placeholder1, placeholder2, placeholder3;
placeholder1 = 1;
for( k = 0; k < max; k++)
{
   g[0][k] = placeholder1;
   placeholder1++;
}

placeholder2 = 1;
for( l = 0; l < max; l++)
{
   h[0][l] = placeholder2;
   placeholder2++;
}

placeholder3 = 1;
for( o = 0; o < max; o++)
{
   r[0][o] = placeholder3;
   placeholder3++;
}

// The code below define the second row of each matrix permutation. For the
// permutation p, the second row is the letter that letter i is mapped to in p,
// i.e., p(i) = g[1][i]. The code searches the p array for each letter i and maps
// it to the p[i+1]letter.
int letter1, letter2; // These are the letters being assigned.
letter1 = 1;
for( s = 0; s < max; s++ )
{
   for( m = 0; m < p.length; m++ )
   {
      if ( p[m] == letter1 && m < p.length-1)
         g[1][s] = p[m+1];

      // Handles the case if the letter is at the end of the array.
      if ( p[m] == letter1 && m == p.length-1 )
         g[1][s] = p[0];
   }
   // Handles the case if the letter does not appear in p. For example,
   // if the letter 1 does not appear in p, then the letter 1 is mapped
   // back to one.
   if ( g[1][s] == 0)
      g[1][s] = letter1;
   letter1++;
```

```
    }

    // Similar loop for the matrix permutation h
    letter2 = 1;
    for( t = 0; t < max; t++ )
    {
      for( n = 0; n < q.length; n++)
      {
        if ( q[n] == letter2 && n < q.length-1)
          h[1][t] = q[n+1];
        if ( q[n] == letter2 && n == q.length-1 )
          h[1][t] = q[0];
      }
      if ( h[1][t] == 0)
        h[1][t] = letter2;
      letter2++;
    }

    // The following code computes the product of gh = r.  Recall that
    // r(x) = h(g(x)).
    for( u = 0; u < max; u++ )
    {
      r[1][u]=h[1][g[1][u]-1];
    }
    return r;
  } //ends the multiply method
} //ends class
```

Permutation Inversion Class

```
/************************************************************
* This class takes any permutation p (an integer array) and
* inverts it to pINV (another integer array)
************************************************************/

public class PermutationInversion
{
   public PermutationInversion()
   {
      // Constructor Method
   }

   public static int [] invert(int x[])
   {
      int p[] = x; // p is the permutation whose inverse we seek.
      int pINV[] = new int[p.length]; // pINV is the inverse of p.

      // a permutation is inverted by reversing the order of the cycle.
      // For example, if p = (1, 2, 4, 5, 3), then pINV = (3, 5, 4, 2, 1).
      for (int i = 0; i < p.length; i++)
         pINV[i] = p[p.length-(i+1)];

      return pINV;
   } // end invert method
} // ends class
```

Permutation Conjugation Class

```
/*****************************************************************************
* This class conjugates two permuations using two different methods.  The first method
* used the definition of conjugation, which is p^m = (mINV)*p*m, where * is the group
* binary operation.  The second method uses a theorem proven for any two permutations.
* This theorem states that the conjugate of p by m, or p^m, is found by taking each letter
* in p and replacing it with its image in m.  For example, if p = (1,3,2,4) and m = (1,2),
* then p^m = (2,3,1,4) since m(1) = 2, m(2) = 1, m(3) = 3, and m(4) = 4.
*****************************************************************************/

public class PermutationConjugation
{
   public PermutationConjugation()
   {
      //Constuctor Method
   }

   public static int [][] conjugate1(int x[], int y[])
   {
      int p[] = x;  // permutation being operated on
      int m[] = y;  // permutation operator
      int s[], mINV[];
      int q[][];   // conjugate result

      // The following code conjugate p by m using the definition
      // p^m = mINV*p*m = s*m = q.
      mINV = PermutationInversion.invert(m);
      //s = PermutationMultiplication.multiply(mINV,p);
      q = PermutationMultiplication.multiply(mINV,p);

      return q;

   } // ends conjugate1 method

   // The conjugation method below finds the conjugate using the theorem stated above.
   public static int[] conjugate2(int x[], int y[])
   {
      int p[] = x;  // permutation being operated on
      int m[] = y;  // operating permutation
      int q[] = new int[p.length]; // conjugate result -- conjugation preserves cycle
                        // structure which requires q to equal p in size.
      int letter, lastletter, unmovedletters; // variables for searching p.
      letter = 0;
      // The following code takes each letter in m (except the last one), finds the
      // equivalent letter in p and replaces it with its image in m.
      for( int i = 0; i < m.length-1; i++ )
      {
         letter = m[i];
         for( int j = 0; j < p.length; j++ )
         {
            if( p[j] == letter )
               q[j] = m[i+1];
         }
```

```
    }
    // The code below considers the last letter in m.
    lastletter = m[m.length-1];
    for( int k = 0; k < p.length; k++ )
    {
      if( p[k] ==  lastletter )
        q[k] = m[0];
    }
    // The following code considers all letter in p that are not moved by m.
    unmovedletters = 0;
    for( int l = 0; l < p.length; l++ )
    {
      if( q[l] == unmovedletters )
        q[l] = p[l];
    }
    return q;
  } // ends conjugate2 method
} // ends conjugation class
```

Data Class

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class DataSets
{
  //D1 is the distance matrix for the 10-City Ohio problem
  static int [][] D1 = { {0,24,227,33,121,186,150,161,122,46},
                         {24,0,224,56,118,184,150,159,140,51},
                         {227,224,0,239,106,52,128,71,198,273},
                         {33,56,239,0,141,197,150,172,107,66},
                         {121,118,106,141,0,68,90,43,128,167},
                         {186,184,52,197,68,0,76,25,152,232},
                         {150,150,128,150,90,76,0,66,81,196},
                         {161,159,71,172,43,25,66,0,127,207},
                         {122,140,198,107,128,152,81,127,0,168},
                         {46,51,273,66,167,232,196,207,168,0} };

  static int [][] cost = getData();

  public static int [][] getData()
  {
    String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
    String sourceURL = "jdbc:odbc:MS Access 97 Database";

    int [][] D = new int[48][48];

    int i;
    try
    {
      Class.forName(driverName);
      Connection fileConnection = DriverManager.getConnection(sourceURL, "Admin", "");
      Statement statement = fileConnection.createStatement();
      ResultSet rs = statement.executeQuery("SELECT * from Capitals");


      i = 0;
      boolean more = rs.next();

      while (more)
      {
        D[i][0] = rs.getInt("Field1");
        D[i][1] = rs.getInt("Field2");
        D[i][2] = rs.getInt("Field3");
        D[i][3] = rs.getInt("Field4");
        D[i][4] = rs.getInt("Field5");
        D[i][5] = rs.getInt("Field6");
        D[i][6] = rs.getInt("Field7");
        D[i][7] = rs.getInt("Field8");
        D[i][8] = rs.getInt("Field9");
        D[i][9] = rs.getInt("Field10");
        D[i][10] = rs.getInt("Field11");
```

```
D[i][11] = rs.getInt("Field12");
D[i][12] = rs.getInt("Field13");
D[i][13] = rs.getInt("Field14");
D[i][14] = rs.getInt("Field15");
D[i][15] = rs.getInt("Field16");
D[i][16] = rs.getInt("Field17");
D[i][17] = rs.getInt("Field18");
D[i][18] = rs.getInt("Field19");
D[i][19] = rs.getInt("Field20");
D[i][20] = rs.getInt("Field21");
D[i][21] = rs.getInt("Field22");
D[i][22] = rs.getInt("Field23");
D[i][23] = rs.getInt("Field24");
D[i][24] = rs.getInt("Field25");
D[i][25] = rs.getInt("Field26");

D[i][26] = rs.getInt("Field27");
D[i][27] = rs.getInt("Field28");
D[i][28] = rs.getInt("Field29");
D[i][29] = rs.getInt("Field30");
D[i][30] = rs.getInt("Field31");
D[i][31] = rs.getInt("Field32");
D[i][32] = rs.getInt("Field33");
D[i][33] = rs.getInt("Field34");
D[i][34] = rs.getInt("Field35");
D[i][35] = rs.getInt("Field36");
D[i][36] = rs.getInt("Field37");
D[i][37] = rs.getInt("Field38");
D[i][38] = rs.getInt("Field39");
D[i][39] = rs.getInt("Field40");
D[i][40] = rs.getInt("Field41");
D[i][41] = rs.getInt("Field42");
D[i][42] = rs.getInt("Field43");
D[i][43] = rs.getInt("Field44");
D[i][44] = rs.getInt("Field45");
D[i][45] = rs.getInt("Field46");
D[i][46] = rs.getInt("Field47");
D[i][47] = rs.getInt("Field48");

D[i][48] = rs.getInt("Field49");
D[i][49] = rs.getInt("Field50");
D[i][50] = rs.getInt("Field51");
D[i][51] = rs.getInt("Field52");

D[i][52] = rs.getInt("Field53");
D[i][53] = rs.getInt("Field54");
D[i][54] = rs.getInt("Field55");
D[i][55] = rs.getInt("Field56");
D[i][56] = rs.getInt("Field57");
D[i][57] = rs.getInt("Field58");
D[i][58] = rs.getInt("Field59");
D[i][59] = rs.getInt("Field60");
D[i][60] = rs.getInt("Field61");
D[i][61] = rs.getInt("Field62");
```

```
D[i][62] = rs.getInt("Field63");
D[i][63] = rs.getInt("Field64");
D[i][64] = rs.getInt("Field65");
D[i][65] = rs.getInt("Field66");
D[i][66] = rs.getInt("Field67");
D[i][67] = rs.getInt("Field68");
D[i][68] = rs.getInt("Field69");
D[i][69] = rs.getInt("Field70");
D[i][70] = rs.getInt("Field71");
D[i][71] = rs.getInt("Field72");
D[i][72] = rs.getInt("Field73");
D[i][73] = rs.getInt("Field74");
D[i][74] = rs.getInt("Field75");
D[i][75] = rs.getInt("Field76");
D[i][76] = rs.getInt("Field77");
D[i][77] = rs.getInt("Field78");
D[i][78] = rs.getInt("Field79");
D[i][79] = rs.getInt("Field80");
D[i][80] = rs.getInt("Field81");
D[i][81] = rs.getInt("Field82");
D[i][82] = rs.getInt("Field83");
D[i][83] = rs.getInt("Field84");
D[i][84] = rs.getInt("Field85");
D[i][85] = rs.getInt("Field86");
D[i][86] = rs.getInt("Field87");
D[i][87] = rs.getInt("Field88");
D[i][88] = rs.getInt("Field89");
D[i][89] = rs.getInt("Field90");
D[i][90] = rs.getInt("Field91");
D[i][91] = rs.getInt("Field92");
D[i][92] = rs.getInt("Field93");
D[i][93] = rs.getInt("Field94");
D[i][94] = rs.getInt("Field95");
D[i][95] = rs.getInt("Field96");
D[i][96] = rs.getInt("Field97");
D[i][97] = rs.getInt("Field98");
D[i][98] = rs.getInt("Field99");
D[i][99] = rs.getInt("Field100");
D[i][100] = rs.getInt("Field101");
D[i][101] = rs.getInt("Field102");
D[i][102] = rs.getInt("Field103");
D[i][103] = rs.getInt("Field104");
D[i][104] = rs.getInt("Field105");
D[i][105] = rs.getInt("Field106");
D[i][106] = rs.getInt("Field107");
D[i][107] = rs.getInt("Field108");
D[i][108] = rs.getInt("Field109");
D[i][109] = rs.getInt("Field110");
D[i][110] = rs.getInt("Field111");
D[i][111] = rs.getInt("Field112");
D[i][112] = rs.getInt("Field113");
D[i][113] = rs.getInt("Field114");
D[i][114] = rs.getInt("Field115");
D[i][115] = rs.getInt("Field116");
```

```
            D[i][116] = rs.getInt("Field117");
            D[i][117] = rs.getInt("Field118");
            D[i][118] = rs.getInt("Field119");
            D[i][119] = rs.getInt("Field120");
            D[i][120] = rs.getInt("Field121");
            D[i][121] = rs.getInt("Field122");
            D[i][122] = rs.getInt("Field123");
            D[i][123] = rs.getInt("Field124");
            D[i][124] = rs.getInt("Field125");
            D[i][125] = rs.getInt("Field126");
            D[i][126] = rs.getInt("Field127");
            D[i][127] = rs.getInt("Field128");
            D[i][128] = rs.getInt("Field129");
            D[i][129] = rs.getInt("Field130");

            D[i][130] = rs.getInt("Field131");
            D[i][131] = rs.getInt("Field132");
            D[i][132] = rs.getInt("Field133");
            D[i][133] = rs.getInt("Field134");
            D[i][134] = rs.getInt("Field135");
            D[i][135] = rs.getInt("Field136");
            D[i][136] = rs.getInt("Field137");
            D[i][137] = rs.getInt("Field138");
            D[i][138] = rs.getInt("Field139");
            D[i][139] = rs.getInt("Field140");
            D[i][140] = rs.getInt("Field141");
            D[i][141] = rs.getInt("Field142");
            D[i][142] = rs.getInt("Field143");
            D[i][143] = rs.getInt("Field144");
            D[i][144] = rs.getInt("Field145");
            D[i][145] = rs.getInt("Field146");
            D[i][146] = rs.getInt("Field147");
            D[i][147] = rs.getInt("Field148");
            D[i][148] = rs.getInt("Field149");
            D[i][149] = rs.getInt("Field150");
            i++;
            more = rs.next();
          }
        }
        catch( ClassNotFoundException cnfe)
        {
          System.err.println("Error Loading " + driverName);
        }

        catch( SQLException sqle)
        {
          System.err.println(sqle);
        }
        //for( int j = 0; j < D2.length; j++)
          //System.out.println( D2[j][0]+" "+D2[j][1]+" "+D2[j][2]+" "+D2[j][3]+" "+D2[j][4]);
        return D;
      }
    } //end class
```

Appendix B. TSP Formulation

B.1 Introduction

The TSP class of problems are modeled mathematically as binary integer programs. This appendix provides a formulation for the TSP, mTSP, mTSPTW, and multiple depot mTSPTW (md-mTSPTW).

B.2 The Traveling Salesman Problem

Recalling the classical TSP, an agent wishes to leave his base location and visit several customer locations exactly once before returning home. The agent seeks the tour that optimizes some objective. Mathematically, suppose we have a network of n nodes and a arcs connecting these nodes. Further, each arc has an associated cost (benefit), c_{ij}, for going from node i to node j. We assume $c_{ij} = 0$ when $i = j$ and either $c_{ij} = c_{ji}$ (symmetric) or $c_{ij} \neq c_{ji}$ (asymmetric). Also, let each network arc have an associated variable x_{ij} such that

$$x_{ij} = \begin{cases} 1 & \text{if the arc } i \text{ to } j \text{ is taken} \\ 0 & \text{otherwise} \end{cases}.$$

The TSP becomes a problem of selecting a minimum cost (maximum benefit) tour, i.e., a collection of x_{ij}, over all n nodes beginning and ending at an origin, say node 1. Clearly this equates to deciding which $x_{ij} = 1$ and which $x_{ij} = 0$. The mathematical formulation that produces an optimal tour defines the objective function as

$$\text{Minimize } \mathbf{CX} = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

where **C** is the cost matrix whose i, j^{th} entry is c_{ij} and **X** is the arc matrix whose i, j^{th} entry is 0 or 1. Since each node must be visited exactly once, there must be only one arc, x_{ij}, beginning at node i and only one arc, x_{ij}, ending at node j for all n nodes. This produces the set of constraints

$$\sum_{i=1}^{n} x_{ij} = 1 \text{ for each } j = 1, \ldots, n \text{ and } j \neq i$$

(Ensures exactly one arc, x_{ij}, ends at node j)

$$\sum_{j=1}^{n} x_{ij} = 1 \text{ for each } i = 1, \ldots, n \text{ and } i \neq j \qquad (1)$$

(Ensures exactly one arc, x_{ij}, begins at node i)

$$x_{ij} \in \{0,1\} \ \forall i, j .$$

The assignment constraints (1) do not eliminate subtours. For example, consider a five node TSP. It is possible that the above formulation would yield the solution $x_{12} = x_{21} = x_{34} = x_{45} = x_{53} = 1$, which contains the subtours $1 \rightarrow 2 \rightarrow 1$ and $3 \rightarrow 4 \rightarrow 5 \rightarrow 3$. Hence, we need a second set of constraints to eliminate any possible subtours. Bodin *et al.* (1983) provide three different ways to represent subtour breaking constraints:

$$\sum_{i \in Q} \sum_{j \notin Q} x_{ij} \geq 1 \text{ for every nonempty proper subset } Q \text{ of the set of nodes } \{1, 2, \ldots, n\}; \qquad (2a)$$

$$\sum_{i \in R} \sum_{j \in R} x_{ij} \leq |R| - 1 \text{ for every nonempty subset } R \text{ of the set } \{2, 3, \ldots, n\}; \qquad (2b)$$

$$y_i - y_j + n x_{ij} \leq n - 1 \text{ for } 2 \leq i \neq j \leq n \text{ for some real numbers } y_i. \qquad (2c)$$

Equation (2a) ensures that every nonempty proper subset of network nodes, namely Q, must be connected to the nodes that are not in Q. Equation (2b) ensures that the solution

contains no cycles since a cycle on R nodes contains at least $|R|$ arcs. For equation (2c), let

$$y_i = \begin{cases} k & \text{if node } i \text{ is visited on the } k^{th} \text{ step in a tour} \\ 0 & \text{otherwise} \end{cases}.$$

Therefore, if $x_{ij} = 1$, then the subtour breaking constraint becomes $k-(k+1)+n \le n-1$, and if $x_{ij} = 0$, then we have $y_i - y_j \le n-1$.

Representations (2a) and (2b) require 2^n constraints while (2c) only requires $n^2 - 3n + 2$ constraints. The sheer number of subtour breaking constraints demonstrates the computational complexity of the TSP formulation. For example, a simple ten node TSP formulation requires $2^{10} = 1024$ subtour breaking constraints if (2a) or (2b) is used and 72 subtour breaking constraints if (2c) is used.

B.3 The Multiple Traveling Salesman Problem

Suppose multiple salesmen (i.e., m of them) are available to travel the network of n nodes, then we have the mTSP. Assuming all m salesmen depart from the same origin (depot) gives a formulation nearly identical to the TSP. The only required change in the TSP formulation for the mTSP occurs in the assignment constraints (1) of section B.2. The mTSP requires that m arcs begin and terminate at the origin (depot) assumed as node 1. Hence, the assignment constraints (1) for the mTSP are

$$\sum_{i=1}^{n} x_{ij} = \begin{cases} m & \text{for } j = 1 \\ 1 & \text{for } j = 2,\ldots,n \end{cases}; j \neq i \text{ and}$$

$$\sum_{j=1}^{n} x_{ij} = \begin{cases} m & \text{for } i = 1 \\ 1 & \text{for } i = 2,\ldots,n \end{cases}; i \neq j.$$

As with the TSP, the mTSP formulation is constrained by equation (2).

B.4 The Multiple Traveling Salesman Problem with Time Windows

Now suppose, in addition to m salesmen, that all or some nodes of the network must be visited within some specified interval of time. (As an illustration, suppose C-5 aircraft arrivals to Travis AFB are permitted only between 0800 and 2000.) This is the mTSP with time windows (mTSPTW).

The mathematical formulation of the mTSPTW is identical to the mTSP formulation discussed in section B.3 with an additional set of time window constraints. Let s_i be the service time at node i, t_{ij} be the travel time from node i to node j, and a_j be the arrival time at node j. Further, let e_j be the earliest arrival time to node j and let l_j be the no later than arrival time. Ryer's (1999) nonlinear representation of time window constraints is

$$a_j = \sum_{i=1}^{n} (a_i + s_i + t_{ij}) x_{ij} \quad \text{for } j = 1, \ldots, n;$$

$$a_1 = 0;$$

$$e_j \le a_j \le l_j \quad \text{for } j = 2, \ldots, n.$$

This representation is intuitive since some $x_{ij} = 1$ for each j. This infers that a_j is the sum of the arrival time to node i, a_i, the service time at node i, s_i, and the travel time from node i to node j, t_{ij}.

Bodin *et al.* (1983) provides the following linear representation for time window constraints:

$$\left.\begin{aligned} a_j &\geq (a_i + s_i + t_{ij}) - M(1 - x_{ij}) \\ a_j &\leq (a_i + s_i + t_{ij}) + M(1 - x_{ij}) \end{aligned}\right\} \text{ for } \forall\, i, j.$$

Note when $x_{ij} = 1$, the previous arrival and service time, a_i and s_i, and the travel time between nodes, t_{ij}, determines a_j; however, when $x_{ij} = 0$, the constraints are non-binding for some large positive M.

B.5 The Multiple Depot Multiple Traveling Salesman Problem with Time Window

The final level of complexity is the addition of multiple depots to the mTSPTW. This problem is the multiple depot mTSPTW or md-mTSPTW. The addition of multiple depots is important to AMC applications since AMC relies on several bases (depots) to meet its strategic airlift requirements.

As with other additions to the classical TSP, the md-mTSPTW is formulated with only a few minor changes in the assignment constraints (constraints (1) in section B.2) and subtour breaking constraints (constraints (2) in section B.2). Let d be the number of depots in the problem network, and let m_y be the number of salesmen at depot y. Then the assignment constraints for the md-mTSPTW become

$$\sum_{i=1}^{n} x_{ij} = \begin{cases} m_j & \text{for } j = 1, \ldots, d \\ 1 & \text{for } j = d+1, \ldots, n \end{cases}; j \neq i \text{ and}$$

$$\sum_{j=1}^{n} x_{ij} = \begin{cases} m_i & \text{for } i = 1, \ldots, d \\ 1 & \text{for } i = d+1, \ldots, n \end{cases}; i \neq j,$$

and the subtour breaking constraints become

$$\sum_{i \in Q} \sum_{j \notin Q} x_{ij} \geq 1 \text{ for every nonempty proper subset } Q \text{ of the set of nodes } \{1, 2, \ldots, d\}.$$

$$\sum_{i \in R} \sum_{j \in R} x_{ij} \leq |R| - 1 \text{ for every nonempty subset } R \text{ of the set } \{d+1, d+2, \ldots, n\}.$$

$$y_i - y_j + nx_{ij} \leq n-1 \text{ for } d \leq i \neq j \leq n \text{ for some real numbers } y_i.$$

Ryer (1999) and Carlton (1995) extend this formulation to the VRP and PDP. Ryer discusses additional formulation constraints faced by AMC such as route length restrictions, crew availability, route or airspace restrictions, winds, and air refueling capability.

Bibliography

Baker, E.K., and J.R. Schaffer. "Solution Improvement Heurisitcs for the Vehicle Routing and Scheduling Problem," *American Journal of Mathematical and Management Sciences, 16*: 261-300 (February 1986).

Battiti, R., R., and G. Tecchiolli. "The Reactive Tabu Search," *ORSA Journal on Computing, 6*: 126-140 (1994).

Bodin, Lawrence, Bruce Golden, A. Assad, and M. Ball. "Routing and Scheduling of Vehicles and Crews; The State of the Art," *Computers and Operations Research, 10*: (1983).

Carlton, William B. *A Tabu Search to the General Vehicle Routing Problem.* Ph.D. Dissertation. The University of Texas at Austin, Austin TX, 1995.

Carlton, William B., and J. Wesley Barnes. "A note on hashing functions and tabu search algorithms," *European Journal of Operational Research,106*: 237-239 (1996).

Colletti, Bruce W. *Group Theory and Metaheuristics.* Ph.D. dissertation. The University of Texas at Austin, Austin TX, 1999.

Colletti, Bruce W. and J. Wesley Barnes. "Group Theory and Metaheuristic Search Neighborhoods." Graduate Program in Operations Research and Industrial Engineering. The University of Texas at Austin, Austin TX, 15 March 1999.

Desrochers, M., J. Desrosiers, and M. Solomon. "A New Optimization Algorithm for the Vehicle Routing Problems with Time Windows," *Operations Research*, 40: 342-353 (1992).

Fraleigh, John B. *A First Course in Abstract Algebra.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1994.

Glover, Fred. "Tabu Search: A Tutorial," *Interfaces, 20*: 74 -94, 1990.

Glover, Fred and M. Laguna. *Tabu Search.* Boston: Kluwer Academic Publishers, 1997.

Harder, Robert. *A Java Universal Vehicle Router in Support of Routing Unmanned Aerial Vehicles.* MS thesis, AFIT/GOA/ENS/00M-15. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000.

Herstein, I.N. *Topics in Algebra.* Waltham, Massachusetts: Xerox College Publishing, 1964.

Laporte, Gilbert. "The Traveling Salesman Problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, 59: 231-247 (1992a).

Laporte, Gilbert. "The Vehicle Routing Problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, 59: 345-358 (1992b).

Moore, James T. Ohio 10-City Traveling Salesman Problem. Department of Operational Sciences, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (1999).

Nemhauser, George L. and L. Wolsey. *Interger and Combinatorial Optimization*. New York: John Wiley & Sons, 1988.

Osman, I.H. "Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem," *Annals of Operations Research*, 41: 421-451 (1993).

Reinelt, Gerhard. *TSPLIB*. Institut für Angewandte Mathematik, Univeristät Heidelberg, Germany. ftp://ftp.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html. 22 January 2000.

Rotman, Joseph J. *An Introduction to the Theory of Groups*. Newton, MA: Allyn and Bacon, Inc., 1984

Ryan, J L., T. G. Bailey, J. T. Moore, and W. B. Carlton. "Unmanned Aerial Vehicle (UAV) Route Selection Using Reactive Tabu Search," *Military Operations Research*, V4 N3: 5-24 (1999).

Ryer, David M. *Implementation of the Metaheuristic Tabu Search in the Route Selection for Mobility Analysis Support System*. MS Thesis, AFIT/GOA/ENS/99M-07. Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1999.

Schönert, Martin, et. al. *GAP--Groups, Algorithms, and Programming*. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische. Hocheschule, Aachen, Germany, fifth edition. http://www-gap.dcs.st-andrews.ac.uk/~gap/. 25 September 1999.

Woodruff, D., and E. Zemel. "Hashing Vectors for Tabu Search," *Annals of Operations Research, Vol. 41*: 123-137 (1993).

Vita

First Lieutenant Shane N. Hall was born 1 July 1973 in Show Low, Arizona to Harry Lynn and Geraldine Hall. After graduating from Show Low High School in May 1991, he attended Brigham Young University and received a Bachelor of Science degree in Mathematics and his commission in April 1997. While at Brigham Young University, he married Camalee R. Hall in April 1995. After receiving his commission, Shane worked in the 96th Communications Group at Eglin AFB, Florida until August 1998 when he entered the Graduate School of Engineering and Management, Air Force Institute of Technology, Wright-Patterson AFB, Ohio.